VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

OpenGL prohlížeč naměřených prostorových dat.

OpenGL Viewer for 3D Measured Data.

2016 Bc. Lukáš Gaál

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Lukáš Gaál**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: OpenGL prohlížeč naměřených prostorových dat.
OpenGL Viewer for 3D Measured Data

Jazyk vypracování: čeština

Zásady pro vypracování:

Navrhněte a upravte OpenGL prohlížeč prostorových dat, aby bylo možno načítat 3D data složená z bodů, přímek a polygonů a aby bylo možno data vizualizovat a pohybovat se v naměřeném prostoru ve všech osách, měnit měřítka v jednotlivých osách a měnit nastavení barev zobrazených dat.

1. Seznamte se s dostupnými prohlížeči pro mračna bodů a porovnejte jejich vlastnosti.
2. Navrhněte základní koncepci prohlížeče, způsob jeho ovládání a pohyb v prostoru. Zvolte formát načítaných dat.
3. Vyberte z možností OpenGL ty techniky, které budou vhodné pro realizaci prohlížeče.
4. Navrhněte a implementujte prohlížeč tak, aby umožňoval načítání více datových souborů s celkovým počtem bodů v řádech miliónů. Při implementaci dbejte na přenositelnost kódu mezi operačními systémy.
5. Implementujte v programu možnost výběru konkrétních bodů a měření vzdáleností.
6. Ověřte funkcionalitu programu, jeho stabilitu a nároky na paměť a procesor.

Seznam doporučené odborné literatury:

[1] Tom Davis, Dave Shreiner, Mason Woo, Jackie Neider, OpenGL - Průvodce programátora, Computer Press, EAN 9788025112755
[2] http://www.opengl.org
[3] http://www.pointclouds.org

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Petr Olivka, Ph.D.**

Datum zadání: 01.09.2014
Datum odevzdání: 29.04.2016

VYSOKÁ ŠKOLA BÁŇSKÁ-TECHNICKÁ UNIVERZITA * Fakulta elektrotechniky a informatiky * OSTRAVA

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 29. dubna 2016 .....

Rád bych na tomto místě poděkoval všem, kteří mi s prací pomohli. Zejména Ing. Petru Olivkovi Ph.D. za vedení této diplomové práce, jeho rady a čas. Bez nich by tato práce nevznikla. Také bych zde chtěl poděkovat své rodině, která mi byla oporou při tvorbě práce.

Abstrakt

Ibis PCD Viewer je počítačový program, který vznikl z důvodu potřeby pohodlně prohlížet naměřená prostorová data. Mezi jeho přednosti patří jednoduché ovládání pomocí myši a klávesnice a široké spektrum nastavení pozice kamery a zobrazení dat během virtuálního procházení scény. Program pro svou činnost využívá možnosti moderních akcelerovaných grafických karet a zároveň si zachovává zpětnou kompabilitu se staršími kartami pomocí knihovny OpenGL. Díky využití pouze multiplatformních knihoven není uživatel programu vázán na konkrétní operační systém.

Klíčová slova: prohlížeč prostorových dat, pcd, OpenGL

Abstract

Ibis PCD Viewer is a computer program, which was created for necessity of convenient viewing the measured spatial data. Its advantages include simple controlling via the mouse and keyboard. Further advantage is wide spectrum of camera and data representation settings during scene virtual browsing. It utilizes the capabilities of modern accelerated graphics cards for its functionality and it is as well backward compatible with older cards thanks to OpenGL library. End user is not bound to a particular computer operating system thanks to utilizing only cross-platform libraries.

Key Words: point cloud viewer, spatial data viewer, pcd, OpenGL

Obsah

Seznam použitých zkratek a symbolů

2D	–	dvoudimenzionální prostor
3D	–	trojdimenzionální prostor
API	–	Application Interface
AGL	–	Apple Graphics Library
CPU	–	Central Processing Unit
FreeGLUT	–	free GL utility toolkit
GLEW	–	GL extension wrangler
GLM	–	OpenGL Mathematics
GLUT	–	GL utility toolkit
GPU	–	Graphic Processing Unit
GUI	–	Graphical User Interface
LIDAR	–	Light Detection And Ranging
MV	–	Model View
MVP	–	Model View Projection
OpenGL	–	Open Graphical Library
OS	–	operační systém
PCD	–	Point Cloud Data
PCL	–	Point Cloud Library
RAM	–	Random Access Memory
SDL	–	Simple DirectMedia Layer
WGL	–	Windows Graphics Library

Seznam obrázků

Seznam tabulek

Seznam výpisů zdrojového kódu

1 Úvod

Zadání diplomové práce (dále DP) jsem si zvolil, abych pochopil principy a omezení, které musí každé programátor řešit při tvorbě aplikace pracující s počítačovou grafikou. Tato diplomová práce se zabývá tvorbou počítačového programu pro zobrazení 3D naměřených prostorových dat, a to konkrétně velkého množství jednotlivých bodů soustředěných v mračnech. 3D data se nejčastěji získávají z technického zařízení, které měří vzdálenost okolních objektů a překážek pomocí laseru. Data se dají také získat za pomoci snímání prostoru více kamerami a následném zpracování vhodným programem (např. Kinect). Měření vzdálenosti je dosaženo na základě měření časového rozdílu mezi časem vyslání a časem přijmutí odraženého laserového paprsku. Tato metoda dálkového průzkumu vzdálenosti bývá označována jako LIDAR.

Ze zadání DP plynou některé hlavní požadavky na program a jeho hlavní rysy. Program umožní plynule zobrazovat a virtuálně procházet scénu tvořenou mračny v řádech milionů bodů. K tomu bude využívat technologii OpenGL. Aktuální pozice kamery a její natočení v souřadné soustavě bude intuitivně zobrazena. Program musí být schopen vybrat jednotlivé body interaktivně pomocí myši a zobrazit o nich základní informace. Zejména důležité jsou informace o souřadnicích bodu nebo vzdálenost mezi vybranými body nebo úhly, které svírají jejich vektory. V programu bude možné barevné odlišení jednotlivých mračen pomocí konfiguračních souborů, parametru příkazového řádku nebo interaktivně přímo v programu pomocí klávesnice. Dalším požadavkem na program je jeho přenositelnost mezi počítačovými platformami. Stejná verze programu proto musí beze změn fungovat na počítačích s různými operačními systémy, zejména na OS Linux a Windows. Hlavním předpokladem pro ovládání programu je využití dvoutlačítkové myši s kolečkem a plnohodnotné klávesnice.

Spolu se zadáním DP jsem dostal i kostru funkčního prototypu existujícího prohlížeče vytvořeného na katedře informatiky. Prohlížeč pro vykreslování používal zastaralých a pomalých funkcí pro vykreslování scény. Z existujícího projektu jsem zachoval pouze kostru základních tříd a některé agregace mezi nimi. Většina kódu byla od začátku přepsána, zejména výpočty pro posun kamery v prostoru. V zadaném prototypu prohlížeče se uživatel nemohl dostat do některých pozic pro prohlížení. Podobným problémem trpí například i vestavěný PCD prohlížeč z projektu PCL.

Součástí práce je i zachování podpory běhu programu na starších systémech a původních verzí OpenGL, a to z důvodu neexistujících nových ovladačů pro některé grafické karty nebo některé počítačové systémy. V tomto případě bývá prostředí OpenGL emulováno softwarově a jeho možnosti jsou omezené pouze na základní množinu funkcí.

Vzhledem k tomu, že program musí splňovat požadavek na nezávislost na operačním systému, bylo zapotřebí zvolit vhodnou knihovnu, která dokáže abstrahovat detaily programování pro konkrétní okenní manažery (GNOME, KDE, XFce, Windows Explorer, ...). Pro jednoduchost a rychlost pochopení jsem si zvolil knihovnu FreeGLUT, která je k dispozici pro všechny hlavní OS. S knihovnou FreeGLUT dále odpadla potřeba zpracovávat uživatelské vstupy přes klávesnici

a myš pomocí programového rozhraní konkrétního OS. Všechny obsluhy uživatelských vstupů jsou řešeny ve formě tzv. zpětných volání mimo jakoukoli smyčku s obsluhou zpráv.

Obsahem této DP je postupné vysvětlení některých matematických pojmů a operací, které jsou nezbytné pro naprogramování jakékoli 3D aplikace, a to zejména práce s matematickými prvky jako jsou vektory a matice. Postupným skládáním elementárních matematických operací na těchto tělesech jsme schopni dosáhnout umístění objektů a kamery ve scéně a vytvořit tak velmi komplexní obraz scény. Neméně důležitý bude i popis afinních a homogenních souřadnic, spolu s rovnoběžným a středovým promítáním z 3D do 2D.

V textu práce budou popsány některé volně dostupné prohlížeče pro 3D mračna se stručným popisem jejich vlastností. V jedné kapitole bude popsán systém OpenGL – zejména proč vznikl, jak se vyvíjel a jeho krátké srovnání se známým DirectX od MS Windows. V práci se budu dále zabývat paměťovými nároky programu, rychlostí vykreslování a časem potřebným pro některé kritické operace. Část práce bude věnována i programátorské a uživatelské dokumentaci. V programátorské dokumentaci budou popsány některé hlavní metody tříd a společné vazby mezi nimi. Z důvodu velikosti programu zde ale nebude popsáno všechno. Pro tento účel bych čtenáře odkázal ke studiu zdrojových kódů.

2 Knihovna OpenGL a další nezbytné knihovny

Při řešení této práce byly zvoleny knihovny, které mají tu výhodu, že fungují na všech hlavních operačních systémech a v současné době se pracuje na podpoře i těch méně známých nebo na podpoře počítačových platforem s procesory ARM. Jedná se o knihovny aktivně vyvíjené a dobře dokumentované. Kromě OpenGL, GLEW a FreeGLUT, které budou popsány níže, jsem při vývoji s výhodou využil i matematickou knihovnu GLM [6].

2.1 OpenGL

OpenGL poskytuje standardizované programové rozhraní pro práci s 2D a 3D grafikou počítače. OpenGL lze chápat spíše jako standardizovaný předpis než konkrétní implementaci. Tímto je docílena nezávislost standardu na operačním systému a použitém hardware.

Implementace knihovny je nejčastěji dodávána výrobcem grafické karty počítače a je současně instalovaná spolu s ovladači grafické karty. V případě absence grafické karty, můžeme dokonce použít implementace OpenGL (nejznámější Mesa3D), které používají čistě softwarovou emulaci nebo vhodným způsobem volají API operačního systému (OpenGL pro Windows) a simulují chování chybějící grafické karty. Takové emulace ale mají za následek pokles výkonu a většinou silně omezenou funkcionalitu.

Funkce pro vytvoření grafického kontextu pracovního okna aplikace a funkce pro práci s fonty nejsou součástí OpenGL. Tyto funkce jsou součástí aplikačních knihoven konkrétních OS. Pomocí těchto funkcí lze získávat i rozšíření, která jsou definována u nových verzí OpenGL – jmenujme např. WGL pro Microsoft Windows, GLS pro X Window nebo AGL pro Apple OS. Rozšíření jsou dále popsána v kap. 2.1.2.

OpenGL neposkytuje žádné funkce pro práci s ostatním hardwarem počítače ani pro zpracování uživatelských vstupů. Knihovna nabízí nízkoúrovňová volání funkcí pro práci s grafikou, v dnešní době okolo 700 funkcí[1], pomocí kterých jsme schopni postupně sestavit vykreslovanou scénu. Na druhou stranu nám nenabízí možnost snadno definovat 3D objekt např. vesmírné lodi a v jednom volání jej vykreslit do scény. Takovéto složitější tvary musejí být postupně sestaveny ze základních částí a vykreslovány postupně. Tyto objekty jsou nejčastěji sestaveny z čar, trojúhelníků a polygonů. Na ně jsou následně aplikovány barvy, textury a matematické transformace. Pro využití Beziérových a NURBS křivek, transformaci pohledů a projekci kamery slouží knihovna GLU[3], která je součástí každé implementace OpenGL.

OpenGL program se dělí na klientskou a serverovou část. Tímto lze docílit vytvoření scény na jednom počítači a její zobrazení na jiném počítači s využitím počítačové sítě. Pokud je OpenGL program spuštěný pouze na jednom počítači, tak nejčastěji hovoříme o klientské části jako o té, kde probíhají příkazy (CPU), a o serverové části jako o té, kde probíhá samotný grafický výpočet scény (GPU).

2.1.1 Historie a vývoj

V roce 1992 vyšel první standard OpenGL v 1.0. Standard vycházel z konceptu projektu Iris GL, což bylo proprietární API pro práci s grafikou, které používala firma Silicon Graphics. Toto API ale bylo kvůli nedostatečné specifikaci nevhodné pro celosvětové použití. Detailní přehled vývojových verzí je zobrazen v tab. 1. Tabulka verzí byla sestavena ze stránek dokumentace OpenGL[10].

Hlavní změny přinesla verze OpenGL 2.0, která definuje tzv. GLSL jazyk. Tato verze přináší možnost programovatelného vykreslovacího řetězce. V tomto jazyku se píší podprogramy, tzv. shadery, které se spouštějí přímo na grafické kartě. Shadery mají syntaxi velice podobnou jazyku *C*. GLSL obsahuje některá omezení – jako příklad uveďme nemožnost rekurzivního volání funkcí.

Další velkou změnou prošlo OpenGL ve verzi 3.0. Tato verze přinesla rozdělení na *core* a *compatibility* profil. Všechny funkce, které byly označeny jako zastaralé, byly v následné verzi 3.1 a jejím *core* profilu odstraněny. *Compatibility* profil zachovává tyto funkce z důvodu zajištění zpětné kompatibility pro starší programy, které již nemohou být přepsány. Ve skutečnosti však většina implementací OpenGL zachovává zastaralé funkce v *core* profilu a fixní vykreslovací řetězec je vnitřně emulovaný pomocí předprogramovaných shaderů. Na tuto implementaci se však nelze spolehnout a pro nové programy je důrazně doporučeno využívání *core* profilu. Výběr cílového profilu se provádí při inicializaci programu, resp. při překladu shaderů.

OpenGL je navržen jako stavový automat. Na jeho začátku jsou mu pomocí volání procedur nastaveny parametry a vstupní data a ty jsou dále využívány během grafického výpočtu. Když jsou posléze změněny některé parametry (např. pozice kamery), a je znovu zavolaná funkce pro vykreslení, všechny původní stavy jsou při výpočtech použity znovu.

2.1.2 Rozšíření a nové funkce

S každou novou verzí OpenGL se do *core* profilu dostávají nové funkce pro práci s grafikou. Tyto funkce již však bývají zakomponované v předcházejících verzích a jsou dostupné přes systém tzv. rozšíření. Rozšíření mohou dodávat jednotliví výrobci karet nebo skupiny, které se podílí na vývoji OpenGL. Rozšiřující funkce často vznikají z důvodu zpřístupnění možností specifické grafické karty nebo jejího výrobce. Jako další důvod vzniku rozšíření je neaktuálnost knihovních souborů operačního systému pro využití OpenGL, které obsahují nezbytné informace pro linker programu.

Jednotliví výrobci a skupiny mají přidělené svoje zkratky, které slouží jako prefixy názvů funkcí. Jako příklady zkratek zde uvedu NV - NVidia, EXT - rozšíření od více výrobců, ARB - skupina starající se o standard OpenGL (OpenGL Architectural Review Board). Jako příklad rozšíření zde uvedu GL_ARB_point_parameters, které slouží pro definování číselných konstant. Tyto konstanty jsou ve fixním zpracování příkazů použity pro výpočet velikosti bodů v závis-

Rok uvedení	Verze OpenGL	Verze GSLS
1992	1.0	
1997	1.1	
1998	1.2	
1998	1.2.1	
2001	1.3	
2002	1.4	
2003	1.5	
2004	2.0	1.10
2006	2.1	1.20
2008	3.0	1.30
2009	3.1	1.40
2009	3.2	1.50
2010	3.3	3.30
2010	4.0	4.00
2010	4.1	4.10
2011	4.2	4.20
2012	4.3	4.30
2013	4.4	4.40
2014	4.5	4.50

Tabulka 1: Oficiální verze OpenGL, GLSL a jejich rok uvedení

losti na vzdálenosti od středu soustavy. Seznam aktuálních rozšíření lze nalézt na internetových stránkách [11].

2.1.3 Vykreslování

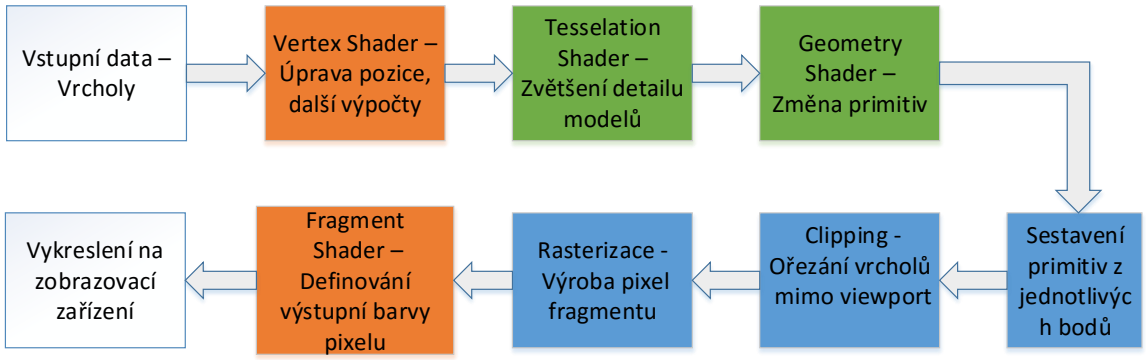
Jelikož OpenGL od verze 3 definuje *core* a *compability* profil, můžeme spatřit dva hlavní rozdíly v přístupech k vykreslování scény. Hlavní princip ovšem zůstává stejný. Všechny objekty jsou definovány pomocí jednotlivých vrcholů a jejich dalších parametrů. Data se dále zpracují na kartě a zapíší se do speciální paměti, tzv. framebufferu. Nad touto pamětí lze následně provádět další operace se scénou (například rozmazání).

Ve starší verzi OpenGL se veškeré vykreslování provádělo jedině mezi příkazy `glBegin()` a `glEnd()`. Mezi těmito voláními se definovaly všechny body, barvy, normály, materiály a osvětlení. Dále za pomocí volání funkcí ze skupiny `glDraw*` proběhlo vykreslení do framebufferu. Další verze OpenGL postupně přinesly možnosti definovat skupiny objektů (display lists) pro vrcholy, barvy atd., a ty vykreslit v jediném volání. Toto vylepšení přineslo úsporu přenosu potřebných dat mezi paměti počítače a grafickou kartou při každém vykreslení scény. Další možností uložení je vše definovat v jediném paměťovém bloku pomocí tzv. *interleaved* polí a následné indexace jednotlivým objektů.

Zrychlení vykreslení dále přinesly *indirect* fukce OpenGL ve formě buffer objektů. Pomocí těchto objetů předáme grafické kartě informace o tom, jak velká data a k jakému účelu je budeme

používat. Ovladač grafické karty si data zkopíruje z paměti počítače do vlastní paměti a další režii si již řeší sám. Tímto způsobem můžeme docílit pouze jediného přenosu dat z RAM do paměti GPU a rapidního zrychlení výpočtu, protože karta má data vždy připravena ve své paměti.

2.1.4 Fixní a programovatelný vykreslovací řetězec



Obrázek 1: Zpracování výpočtu na OpenGL pomocí shaderů

V počátku OpenGL obsahovalo několik modulů, které postupně prováděly výpočet a zpracování dat (rasterizaci). Tyto moduly byly pevně seřazeny do fronty a provádění jejich výpočtu bylo ovlivňováno pouze nastavenými stavy OpenGL. Tento způsob výpočtu je označován jako fixní pipeline (vykreslovací řetězec). Od verze OpenGL 3.1 je fixní model zpracování pouze součástí *compability* profilu. Funkce pro vykreslování pixelů nebo `glBegin` či `glEnd` již taky v *core* profilu nejsou k dispozici. Datové typy, které mohou být použity jako vstupní data pro vykreslovací řetězec, jsou uvedeny v tab. 2. Tabulka byla sestavena na základě údajú [3]. Datové typy, se kterými lze přímo pracovat v jazyce GLSL, jsou uvedeny v tab. 3. Tato tabulka byla sestavena z tabulek [2]. Jazyk GLSL také umožňuje tvorbu nových datových struktur.

Od Verze OpenGL 2.0 máme možnost ovlivňovat grafický výpočet krátkými programy (shadery). V těchto shaderech ovlivňujeme zpracování vrcholů, fragmentů, barev. Grafická primitiva jako vrcholy, trojúhelníky a polygony zde můžeme zde měnit, skartovat nebo je lze naopak dynamicky generovat během výpočtu. Pro každou z těchto kategorií úprav existuje vlastní kategorie shaderů – viz obr. 1. Obrázek byl sestaven na základě informací [2].

Vertex shadery slouží pro zpracování grafických primitiv. Fragment shadery definují výslednou barvu grafického fragmentu. Tyto dva typy shaderů jsou nutnou a zároveň minimální podmínkou pro správné přeložení programu, který využívá GLSL. OpenGL obsahuje interní kompilátor a linker. GLSL program se proto překládá při každém spuštění OpenGL aplikace. Existuje zde možnost binární GLSL program uložit a při spuštění rychleji načíst. Je zde ovšem

Datový typ OpenGL	ekvivalent v c	Velikost v bytech
GLbyte	char	znaménkový, 1 byte
GLubyte	unsigned char	neznaménkový, 1 byte
GLshort	signed short	znaménkový, 2 byty
GLushort	unsigned short	neznaménkový, 2 byty
GLint	integer	znaménkový, 4 byty
GLuint	unsigned integer	neznaménkový, 4 byty
GLfixed	integer	znaménkový fixní bod, 4 byty
GLfloat	float	4 byty
GLhalf	unsigned short	2 byty
GLdouble	double	8 bytů
GLint(2_10_10_10)	int	komprimovaný signed integer formát, 4 byty
GLuint(2_10_10_10)	int	komprimovaný unsigned integer formát, 4 byty

Tabulka 2: Datové typy, které podporuje OpenGL a jejich ekvivalenty v C

riziko, že při změně hardwarové konfigurace počítače nebude program fungovat. Shaderů můžeme mít více, stejně jako výsledných programů. Za běhu OpenGL aplikace mezi nimi lze přepínat.

Ostatní kategorie shaderů jsou nepovinné. Tesselation shadery slouží pro dynamické generování velkého množství úseček a trojúhelníků během zpracování ostatních grafických primitiv. Tímto způsobem můžeme např. volit mezi detailnějším vykreslením modelu v závislosti na vzdálenosti kamery nebo rychlosti grafické karty.

Geometry shadery slouží pro dynamické generování zcela nových primitiv během výpočtu. Jako vstup přijímá jediné grafické primitivum a na jeho výstupu mohou být zcela jiná, nová nebo žádná primitiva. Jako novinku můžeme označit Compute shadery, které slouží pro složité výpočty prováděné přímo na grafické kartě. S jejich pomocí lze rychle počítat fyzikální změny objektů nebo složité částicové systémy. Dají se přirovnat ke knihovnám CUDA a OpenCL, které také poskytují možnost přímého výpočtu na grafické kartě.

Compute shadery nejsou součástí vykreslovacího řetězce a během příkazů `glDraw*` se nespouštějí. Pro jejich spuštění existují nové příkazy jako `glDispatchCompute()`, `glDispatchComputeIndirect()`.

2.2 GLUT a FreeGLUT

Pro komunikaci s ostatním hardware počítače a pro zpracování událostí GUI musíme používat systémová volání jednotlivých operačních systémů nebo použít univerzální knihovnu. Program využívá knihovnu FreeGLUT [8], která vychází ze specifikace knihovny GLUT. GLUT (OpenGL Utility Toolkit) napsal Mark Kilgard pro rychlejší vývoj a testování aplikací na nejrůznějších operačních systémech. Knihovna ovšem zastarala a její licence neumožňovala další vývoj. Knihovna poskytuje základní práci s okny, kontextovým menu, písmem a zpracování uživatelských vstupů z klávesnice, myši a ostatních vstupních zařízení. Knihovna se nehodí pro sofistikované aplikace s bohatým GUI. Pro takové aplikace je lepší využít knihoven jako Qt nebo SDL. S rozšířenými

Datový typ GLSL	Význam
float	základní datový typ
int	základní datový typ, znaménkový
uint	základní datový typ, neznaménkový
bool	základní datový typ, true - false
vec2, vec3, vec4	vektor dimenze 2,3,4, typ float
ivec2, ivec3, ivec4	vektor dimenze 2,3,4, typ int
uvec2, uvec3, uvec4	vektor dimenze 2,3,4, typ unsigned int
bvec2, bvec3, bvec4	vektor dimenze 2,3,4, typ boolen
mat2, mat3, mat4	čtvercová matice dimenze 2,3,4, typ float
mat2x2, mat2x3, mat2x4 mat3x2, mat3x3, mat3x4 mat4x2, mat4x3, mat4x4	ostatní matice, typ float

Tabulka 3: Základní a složené datové typy v GLSL

možnostmi knihoven ovšem roste složitost programu a jeho velikost. Program využívá knihovnu ve verzi 3.0.

2.3 Glew

Glew [7] je knihovna, jejíž autoři jsou Milan Ikits a Marcelo Magallon. Knihovna umožňuje snadno získávat ukazatele na funkce, které poskytují novou funkcionalitu a rozšíření od jednotlivých výrobců grafických karet. Zároveň ve své implementaci skrývá řešení komplexních problémů spojených s dynamickým načítáním knihoven a získávání jednotlivých funkcí. Takto lze získat ukazatel na funkce, které podporují pouze konkrétní typy grafických karet, a tím získat ještě věrohodnější nebo rychlejší zobrazení scény. Program využívá knihovnu ve verzi 1.11.

2.4 GLM

GLM [6] je v podstatě soubor funkcí, které jsou definovány v hlavičkových souborech *hpp* a tak není problém ji přidat do jakéhokoli projektu, který využívá výpočet s vektory a maticemi. GLM se mimo jiné drží standardu GLSL, tzn. že knihovní datové typy a názvy funkcí se shodují. Toto přináší výhodu při přímém použití s OpenGL, kde se již nemusí datové struktury nijak konvertovat. Program využívá knihovnu ve verzi 0.9.6.

3 Matematický model pro implementaci prohlížeče

Pro sestavení scény s mračny bodů a možnosti se v této scéně virtuálně pohybovat je potřeba matematický model a koncept kamery. Místo pohybu kamery v prostoru se aplikují inverzní matematické transformace na načtené body, které se následně vykreslí na zobrazovací zařízení.

3.1 Afinní a homogenní prostory

Pro popis každého bodu v prostoru a provádění základních transformací, lze použít zápis pomocí tříprvkového vektoru v afinním prostoru:

$$v = (x, y, z). \tag{3.1}$$

Ovšem pro vytvoření perspektivního promítání (tzn. že se stejné objekty jeví různě velké v závislosti na vzdálenosti od kamery) a lineárního posunu, používáme homogenní souřadnicovou soustavu. Homogenizace (změna soustavy z afinní na homogenní) se provede rozšířením původního tříprvkového vektoru na čtyřprvkový:

$$v' = (x, y, z, w) \text{ , kde } w = 1. \tag{3.2}$$

Všechny transformace bodů potom provádíme pomocí násobení vektoru o čtyřech prvcích s maticemi o velikosti 4×4 prvků.

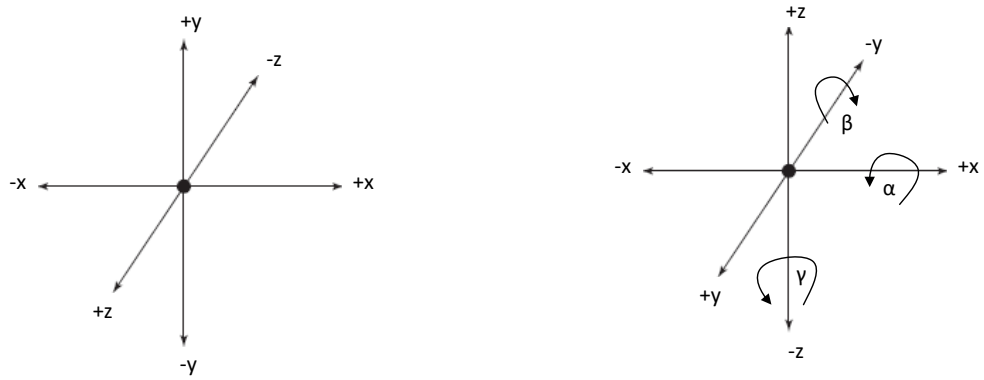
Před mapováním na rozměry zobrazovacího zařízení je nutné provést normalizaci zpět na tříprvkový vektor afinního prostoru:

$$t' = (\frac{x}{w}, \frac{y}{w}, \frac{z}{w}). \tag{3.3}$$

To znamená, že první tři komponenty vektoru homogenní soustavy vydělíme poslední souřadnicí w (předpoklad je, že w je různé od nuly). Ve skutečnosti normalizaci provádí OpenGL před mapováním na zobrazovací zařízení automaticky. Manuálně je tento výpočet prováděn v programu pouze při výpočtu souřadnic během vykreslování textu.

3.2 Zápis matic

Matice z lineární algebry lze zapsat buď řádkově nebo sloupcově orientované. U sloupcově orientovaných matic se vektory zapisují shora dolů a tvoří tak jednotlivé sloupečky. Tato reprezentace je běžně používaná v OpenGL a pracuje s ní taky matematická knihovna GLM. Dále v textu proto budeme předpokládat tuto reprezentaci matic. Na rozdíl od řádkové orientovaných matic se všechny operace, resp. jejich dílčí multiplikace, musí provádět v opačném pořadí. Ve vztahu (3.4)



(a) Souřadnicová soustava OpenGL

(b) Pravotočivá souřadnicová soustava použita v programu

Obrázek 2: Rozdíl v souřadnicové soustavě OpenGL a soustavě použité v programu

lze vidět dosažení stejného výsledku.

$$v' = \begin{cases} (((v \ \times M_1) \times M_2) \times M_3) \text{ [řádkově]} \\ (M_1 \times (M_2 \times (M_3 \times v))) \text{ [sloupcově]} \end{cases} \tag{3.4}$$

3.3 Změna souřadnicového systému

OpenGL neposkytuje žádný pokročilý mechanismus pro jednoduché nastavení scény, resp. nastavení pozice kamery. Přitom abychom nějakou scénu zachytili, musíme nejprve všechny objekty naaranžovat na správná místa. Potom je nutné umístit kameru, nastavit parametry čočky a snímanou scénu zachytit.

Místo pohybu kamery můžeme také posunout všechny objekty scény v opačném směru. Tímto efektem lze docílit vnímání změny kamery v prostoru. Jako příklad lze uvést posun kamery doprava a přiblížení scény objektivem. Takového efektu lze dosáhnout posunutím všech objektů doleva a jejich následným zvětšením.

Každému takovému posunu bodu v prostoru (změně souřadnic) odpovídá určitá lineární transformace A. To znamená, že umístění kamery ve scéně odpovídá určitý počet jednotlivých transformací, které jsou prováděny na všech bodech jednotlivých objektů. Transformace bodu z jedné soustavy do jiné se dá zapsat pomocí maticového násobení:

$$v' = A \times v, \tag{3.5}$$

kde v' jsou souřadnice bodu (vektor) v nové souřadnicové soustavě, v je vektor v původní soustavě a A je transformační matice.

Pro některé jednoduché transformace, jako je posun bodu po jedné ose nebo rovnoměrné škálování, lze transformační matici sestavit přímo. Pro složitější transformace, jako převrácení scény vzhůru nohama, její zkosení a provedení perspektivního promítání, se používá postupné aplikování transformací na jednotlivé body:

$$v' = (A1 \times (A2 \times ...An)) \times v. \tag{3.6}$$

Jednotlivé kroky při násobení matic lze shrnout do jediné transformace:

$$A_k = (A1 \times (A2 \times ...An)). \tag{3.7}$$

Výsledný bod je potom reprezentován vektorem:

$$v' = A_k \times v. \tag{3.8}$$

Jednotlivé dílčí maticové transformace, které vedou k získání výsledné transformační matice jsou zobrazeny na obr. 3. Obrázek byl převzat a upraven [2]. V následující části budou popsány hlavní transformační matice homogenní soustavy. Matice pro dílčí transformace byly převzaty [2].

Translace

$$\mathbf{T} = \begin{pmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.9}$$

Jedná se o elementární transformaci bodu v prostoru, která provádí posun bodu v jednotlivých osách. Za příklad vezměme bod o afinních souřadnicích $v = (0, 0, 0)$. Tento bod rozšíříme do homogenní souřadnicové soustavy $v = (0, 0, 0, 1)$. Homogenní souřadnice takto posunutého bodu lze vyjádřit jako $v' \times T$. Dále předpokládejme jeho posun v ose x o 5 jednotek a v ose y o 10 jednotek a v ose z o 20 jednotek:

$$\mathbf{v'} = \begin{pmatrix} 1 & 0 & 0 & 5 \\ 0 & 1 & 0 & 10 \\ 0 & 0 & 1 & 20 \\ 0 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \times 0 + 0 \times 0 + 0 \times 0 + 5 \times 1 \\ 0 \times 0 + 1 \times 0 + 0 \times 0 + 10 \times 1 \\ 0 \times 0 + 0 \times 0 + 1 \times 0 + 20 \times 1 \\ 0 \times 0 + 0 \times 0 + 0 \times 0 + 1 \times 1 \end{pmatrix} = \begin{pmatrix} 5 \\ 10 \\ 20 \\ 1 \end{pmatrix}. \tag{3.10}$$

Po normalizaci dostaneme souřadnice posunutého bodu $v' = (5, 10, 20)$.

Rotace

Matice pro provedení rotace se dá odvodit s využitím goniometrického počtu, kdy jednotlivé sloupce matice vyjadřují nové bázové vektory prostoru v otočeném souřadném systému. Pro rotace kolem os X, Y, Z o úhly α, β, γ se používají následující matice:

$$\mathrm{R} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & cos(\alpha) & -sin(\alpha) & 0 \\ 0 & sin(\alpha) & cos(\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \begin{pmatrix} cos(\beta) & 0 & -sin(\beta) & 0 \\ 0 & 0 & 0 & 0 \\ sin(\gamma) & 0 & cos(\beta) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \begin{pmatrix} cos(\gamma) & -sin(\gamma) & 0 & 0 \\ sin(\gamma) & cos(\gamma) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{3.11}$$

Příklad bodu $v = (10, 5, 1, 1)$ otočeného o 90° proti směru hodinových ručiček:

$$\mathbf{v}' = \begin{pmatrix} cos(90) & -sin(90) & 0 & 0 \\ sin(90) & cos(90) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 10 \\ 5 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \times 10 + -1 \times 5 + 0 \times 1 + 0 \times 1 \\ 1 \times 10 + 0 \times 5 + 0 \times 1 + 0 \times 1 \\ 0 \times 10 + 0 \times 5 + 1 \times 1 + 0 \times 1 \\ 0 \times 10 + 0 \times 5 + 0 \times 1 + 1 \times 1 \end{pmatrix} = \begin{pmatrix} -5 \\ 10 \\ 1 \\ 1 \end{pmatrix}. \tag{3.12}$$

Škálování

Jedná se o další základní transformaci, která se dá sestavit přímo. Indexy x, y, z slouží ke změně měřítka v odpovídajících osách:

$$\mathbf{S} = \begin{pmatrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{3.13}$$

Příklad čtyřnásobného zvětšení vektoru x, y, z, w:

$$\mathbf{v}' = \begin{pmatrix} 4 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 \\ 0 & 0 & 4 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} = \begin{pmatrix} 4 \times x + 0 \times y + 0 \times z + 0 \times w \\ 0 \times x + 4 \times y + 0 \times z + 0 \times w \\ 0 \times x + 0 \times y + 4 \times z + 0 \times w \\ 0 \times x + 0 \times y + 0 \times z + 1 \times w \end{pmatrix} = \begin{pmatrix} 4x \\ 4y \\ 4z \\ w \end{pmatrix}. \tag{3.14}$$

Modelovací matice

Modelovací matice se používá pro umístění objektu ve scéně. Většinou se skládá z dílčího škálování, rotace a transformace, popř. jejich kombinací.

Všeobecně se pro tuto transformaci užívá název *Model*.

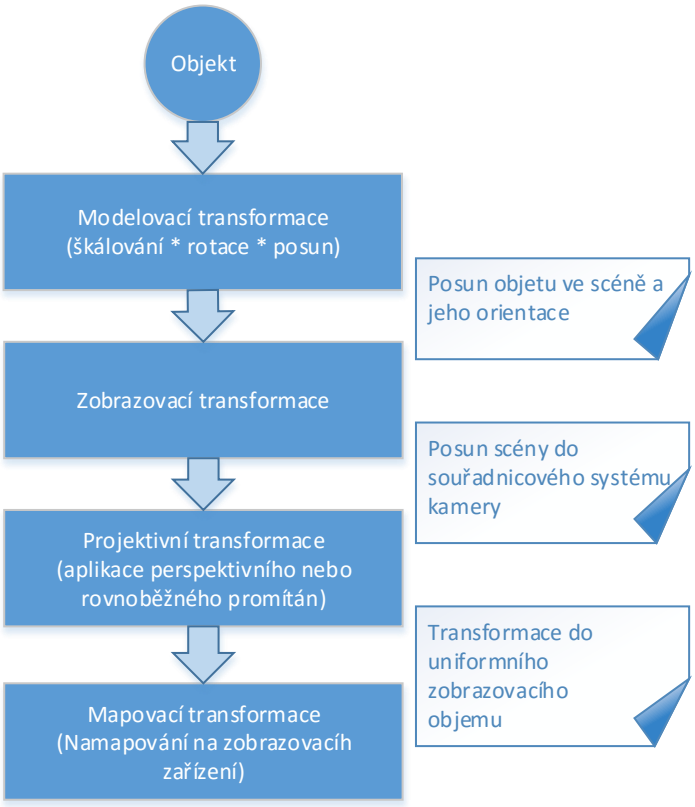
Zobrazovací matice

Zobrazovací matice se používá pro umístění kamery ve scéně, resp. pro posunutí scény. Pro získání takovéto transformační matice lze využít funkci `gml::lookAt()`. Program však tuto funkci nepoužívá a transformační matici sestavuje opět pomocí dílčích rotací a transformací.

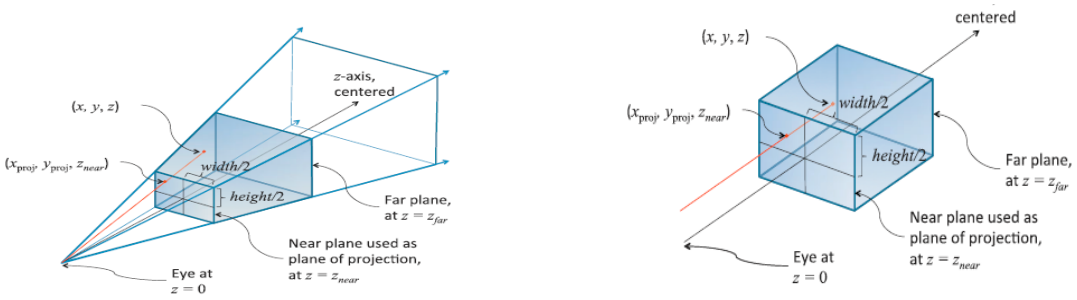
Všeobecně se pro takovou transformaci používá název *View*.

Projektivní matice

$$\mathbf{Persp} = \begin{pmatrix} \frac{z_{bližná}}{šířka/2} & 0 & 0 & 0 \\ 0 & \frac{z_{dálná}}{výška/2} & 0 & 0 \\ 0 & 0 & -\frac{z_{dálná}+z_{bližná}}{z_{dálná}-z_{bližná}} & \frac{2z_{dálná}z_{bližná}}{z_{dálná}-z_{bližná}} \\ 0 & 0 & -1 & 0 \end{pmatrix} \tag{3.15}$$

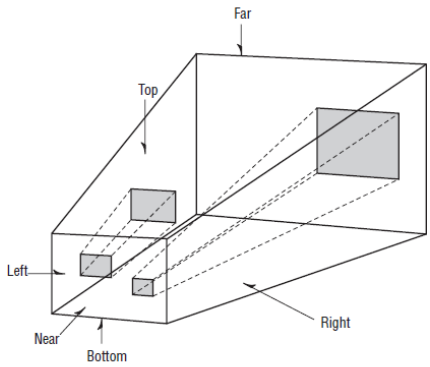


Obrázek 3: Zobrazení složení dílčích transformací



(a) Perspektrivní zobrazovací objem (frustrum) (b) Ortografický zobrazovací objem (frustrum)

Obrázek 4: Rozdíl v použitých projektivních promítáních; obr. převzaty [2].



Obrázek 5: Zmenšení vzdálenějších objektů při perspektivním promítání; obr. převzat [3]

$$\textbf{Orto} = \begin{pmatrix} \frac{1}{\check{s}\acute{i}\check{r}ka/2} & 0 & 0 & 0 \\ 0 & \frac{1}{v\acute{y}\check{s}ka/2} & 0 & 0 \\ 0 & 0 & \frac{-1}{(z_{dálná} - z_{bližná})/2} & -\frac{z_{dálná} + z_{bližná}}{z_{dálná} - z_{bližná}} \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.16}$$

Projektivní transformační matice slouží k dosažení efektu perspektivního nebo rovnoběžného (ortografického) promítání. Rozdíly v těchto promítáních jsou vyobrazeny na obr. 4. U perspektivního promítání se objekty vzdálenější od kamery jeví jako menší – na rozdíl od rovnoběžného promítání. Tato vlastnost je znázorněna na obr. 5. Rovnoběžné promítání se využívá pro vytvoření 2D efektu nebo v různých CAD nástrojích.

Transformace má za úkol umístit všechny body do tzv. jednotkového zobrazovacího objemu (frustrum) mezi roviny promítání – viz obr. 4. Body, které jsou mimo zobrazovací jehlan nebo kvádr OpenGL zahodí. Šířka, výška a hloubka zobrazovacího objemu je touto transformací nastavena na rozměr $[-1, 1]$. Po projekci na bližnou rovinu se využívá z souřadnice k řešení viditelnosti objektů.

Tato matice se ve většině případů vypočítá na začátku a dále se mění pouze v případě změny typu promítání nebo druhu kamery (objektivu). Pro získání této matice lze využít funkci z knihovny GLM jako `glm::perspective()`, `glm::ortho()`. Jako parametry slouží vzdálenost bližší a vzdálenější promítací roviny na ose z, jednotlivé body promítacích rovin, popř. pozorovací úhel, ze kterého se tyto body dopočítají.

Všeobecně se pro takovou transformaci používá název *Projection*.

3.4 Skládání transformací

Všechny předchozí transformace jako rotace, transformace, škálování, projekce a další se dají nahradit jedinou společnou transformací. Taková transformační matice T se dá sestavit z dílčích transformací $T_1, T_2, ...T_n$. Pomocí postupného násobení podle (3.7) lze dojít k výsledné transformační matici.

Tímto postupem jsou v programu získány hlavní transformace, které jsou označovány jako *Model, View, Projection, MV a MVP*. Výsledná transformace *MVP* se získá jako:

$$MVP = Projection \times View \times Model. \tag{3.17}$$

Na všechny body se při výpočtech použije jediná, předem vypočítaná matice a souřadnice bodu se získají jako:

$$v' = MVP \times v. \tag{3.18}$$

Jelikož jsou všechny matice sloupcově orientované, lze výslednou transformaci chápat jako postupné aplikování transformací zprava doleva. Toto odpovídá programovacímu stylu, kdy je každý bod nejprve pomocí transformací umístěn ve scéně, potom je posunut vzhledem k pozici kamery, a nakonec je na něho aplikována projektivní transformace.

Mapování na zobrazovací zařízení

Jako poslední krok při transformaci bodu se provádí normalizace (přechod k afinním souřadnicím) a mapování na výstupní zařízení. Tento krok nebývá součástí předchozích transformací (ani výpočtových shaderů), protože se o něho interně stará OpenGL. Tímto postupem můžeme stejnou scénu v jednom vykreslovacím cyklu vykreslovat na několik různých zobrazovacích zařízení, nebo ji vykreslovat pouze do jejich částí, aniž bychom museli většinu transformačních matic znovu násobit.

4 Popis programátorské části

Program Ibis PCD Viewer [1] se snaží o objektově orientované paradigma při řešení problému návrhu. Některé části kódu nicméně vykazují znaky procedurálního stylu programování. Toto je zapříčiněno procedurálním přístupem OpenGL, které pracuje jako stavový stroj.

4.1 Hlavní objekty

Z důvodu minimalizace zpracovávání výjimek při konstrukci objektů byl zvolen přístup s dvoufázovým konstruováním objektu. To znamená, že se v samotném konstruktoru třídy vyvarujeme alokování prostředků nebo volání metod, u kterých hrozí vyvolání výjimky. V další fázi musí být na takto vytvořeném objektu zavolána metoda `Init()`, ve které se vytváří všechny dynamické prostředky, se kterými třída dále pracuje. V případě chyby tato metoda vrací hodnotu -1 a takovýto objekt by se neměl dále používat. Výhodou je, že nad takovýmto objektem je bezpečné kdykoli zavolat jeho destruktor.

Zobrazení všech hlavních a některých pomocných tříd se statickými funkcemi je znázorněno na diagramu 6. Vstupní funkce programu `main()` se nachází v souboru *entry.cpp*. Zde se provádí pouze inicializace objektu třídy *ScreenManager*, která slouží jako hlavní třída programu, a následně jeho spuštění metodou *run()*.

4.1.1 ScreenManager

Jedná se o základní třídu programu, která je navržena jako jedináček. Třída obsahuje privátní konstruktor a tudíž se nedá vytvořit standardním způsobem. Během inicializace se nastavují základní parametry OpenGL a registrují se nezbytná zpětná volání knihovny pro práci s okny v knihovně FreeGLUT.

Pro práci s kolekcí mračen, informačním panelem a dalšími třídami, které jsou vykreslovány na obrazovku, slouží soukromá kolekce *m_screenEntities*, která uchovává objekty typu *I_InitDraw*. Nad touto kolekcí je v inicializační fázi volána metoda `Init()` a následně při vykreslování metoda `Draw()`.

Třída dále definuje globální proměnnou *USE_GL_3*, pomocí které ostatní třídy rozlišují způsob inicializace a vykreslování – tedy zdali budou použity funkce pouze z *core* profilu nebo bude využíván *compatibility* profil, resp. funkce OpenGL verze 1.0. Tato proměnná odpovídá úspěšně zavedené verzi OpenGL 3.0. Proměnná je nastavena na *true* pouze po úspěšném sestavení vykreslovacího řetězce.

Následuje krátký přehled důležitých metod:

- `getManager()` – statická metoda třídy, která slouží k získání ukazatele na instanci objektu, který je vytvořen v zásobníku během startu programu.

[1] Jméno programu vzniklo spojením názvu původního programu z katedry informatiky *pcd viewer* a vlastní-hokódového označení projektu *projekt Ibis*. Kódové označení odvozené ze zvířecího názvu autor textu používá u všech svých programů.

Obrázek 6: Diagram tříd programu

- `init(argc, argv**)` – metoda, pomocí které se předávají parametry z příkazového řádku k dalšímu zpracování. S využitím externí knihovny je vytvořeno jediné okno programu s nekonečnou smyčkou zpráv. Při otevření okna je také definován mód otevření. Program využívá módy s podporou:
  - GLUT_RGBA – pro definování všech barevných kanálů a průhlednosti.
  - GLUT_ALPHA – pro podporu průhlednosti.
  - GLUT_DEPTH – pro podmíněné vykreslení fragmentu na základě z-souřadnice.

– GLUT_DOUBLE – pro možnost vykreslovat do oddělených barevných bufferů, mezi kterými se přepíná.

Dále jsou v této metodě vytvářeny instance objektů, které jsou určeny k vykreslování na obrazovku nebo jiné pomocné objekty. Součástí metody je dynamické získání ukazatelů na moderní funkce OpenGL. Pro získání těchto funkcí je volána metoda `glewInit()`.

- `initShaders()` – metoda je volána pouze po úspěšném získání ukazatelů na funkce standardu OpenGL v 3.0. Z jednotlivých programovatelných shaderů je zde sestaven kompletní OpenGL program (zobrazovací řetězec).
- `on_display()` – metoda slouží jako zpětné volání při vykreslování okna programu. Na začátku je zde zresetovaná kamera do základní pozice, nastaven viewport pro zobrazovací zařízení a uvolněny OpenGL buffery. V této metodě je vypočítán časový interval od posledního volání metody a ten je předáván při volání metod `Draw()` objektů v kolekci `m_screenEntities`.
- `run()` – tato metoda slouží pro spuštění programu. Měla by být volána pouze po úspěšném volání metody `init()`. V této metodě je vytvořena programová smyčka, ve které se zpracovávají události knihovnou FreeGLUT. K návratu z metody dochází až po ukončení programu nebo zavření okna.

4.1.2 I_InitDraw

Jedná se o třídu, která je využívána jako rozhraní ve smyslu jazyka *C++*. Tato abstraktní třída nabízí funkcionalitu inicializace a vykreslování na obrazovku. Z této abstraktní třídy dědí všechny třídy v programu, které se vykreslují. Třída poskytuje tyto dvě virtuální abstraktní metody, které musí být ve třídě potomka přepsány:

- `virtual bool Init() = 0`,
- `virtual void Draw(int timeMilliseconds) = 0`.

4.1.3 Camera

Camera je druhá nejdůležitější třída programu, která má na starosti výpočet nezbytných transformačních matic. Pomocí svých metod dokáže vracet pouze dílčí transformace nebo kompletní MVP.

Součástí třídy jsou i dvě instance třídy *Floor*, které se vykreslují na základě stavu kamery – tzn. pouze při posunu v jednotlivých osách nebo plochách.

Další vytvářenou třídou je *RotationPoint*, která po vykreslení symbolizuje střed otáčení kamery. Při velmi malé vzdálenosti kamery od středu se střed otáčení nevykresluje. Tyto instance

jsou součástí třídy *Camera*, protože se vykreslují v různých fázích vytváření modelovací transformační matice.

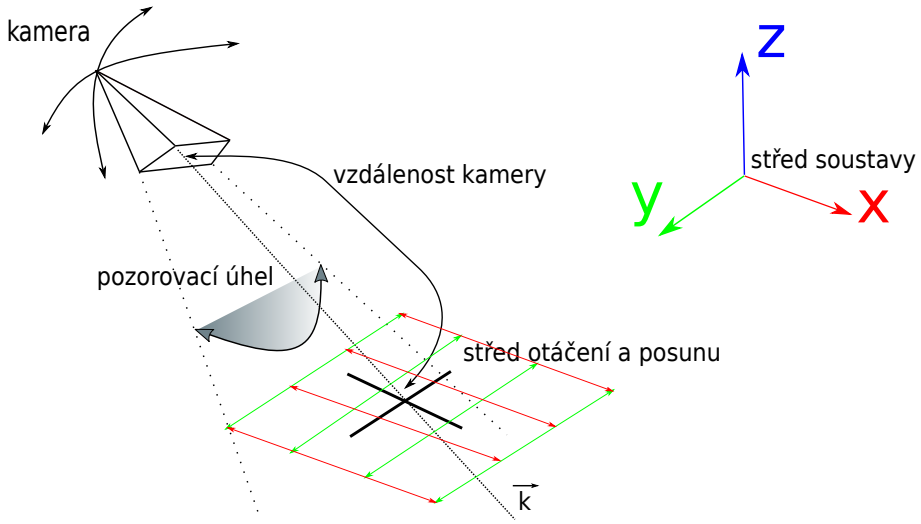
Nedílnou součástí třídy je i enumerační typ *CameraState*, který symbolizuje stav kamery. Jednotlivé stavy jsou:

- `camera_idle` – kamera nemění svůj stav při pohybu myši.
- `camera_zooming` – změna přiblížení.
- `camera_orbiting` – kamera rotuje kolem svého středu otáčení (obr. 7).
- `camera_move_verticaly` – posun ve vertikální rovině.
- `camera_move_horizontal` – posun v horizontální rovině.
- `camera_move_nose_direction` – posun v rovině kamery YZ.
- `camera_change_distance` – přiblížení nebo oddálení od středu otáčení. Minimální hodnota je nastavena na 0.1.
- `camera_move_forward` – posun středu otáčení v ose kamery **k** (obr. 7).
- `camera_move_x` – posun kamery v ose **x**.
- `camera_move_y` – posun kamery v ose **y**, se zanedbáním příčného natočení.
- `camera_move_z` – posun kamery v ose **z**.

Třída kamery také poskytuje funkcionalitu pro úpravu maticových transformací. Pro využívání této možnosti třída disponuje zásobníkem pro dílčí zobrazovací matice *m_view_matrix_stack*. Ostatní třídy, které této funkcionality využívají pro výpočet své modelovací transformace, nesmí porušit aktuální vrchol zásobníku. Proto nejprve uloží stav zásobníku jeho navýšením, a následně na vrcholu vytvoří jednotkovou matici, na kterou jsou aplikovány další transformace. Po dokončení skládání transformací je výsledná matice vyzvednuta a zásobník je obnoven do předchozího stavu. Hlavní funkcionalitu třídy poskytují tyto metody:

- `glm::mat4 getMVPMatrix()` – funkce vrátí spočítanou MVP transformační matici.
- `glm::mat4 getViewMatrix()` – funkce vrátí spočítanou MV transformační matici;
- `camTranslate(glm::vec3 transVec)` – na zásobník aplikuje translační transformaci.
- `camRotate(float degree, glm::vec3 axe)` – na zásobník aplikuje rotaci kolem zvolené osy.
- `camScale(glm::vec3 scaleVec)` – na zásobník aplikuje škálovací transformaci.
- `popViewMatrix()` – vyzvedne výslednou transformační matici ze zásobníku.

- `pushViewMatrix()` – uloží aktuální stav zásobníku. Na nový vrchol je zkopírován poslední stav.
- `loadIdentitiy()` – vrchol zásobníku je přepsán jednotkovou transformační maticí.
- `setState(CameraState param)` – nastaví požadovaný stav kamery.
- `CameraState getState()` – vrátí stav kamery.
- `updatePosition(glm::vec2 delta)` – nejdůležitější metoda této třídy, která aktualizuje modelovací transformační matici o hodnotu *delta* v závislosti na nastaveném stavu kamery.
- `createSnapshot(Snapshot& snap)` – uloží svůj aktuální stav do serializačního objektu.
- `restoreSnapshot(Snapshot& snap)` – obnoví stav kamery ze serializačního objektu.



Obrázek 7: Ovládání kamery v prostoru s naznačeným horizontálním posunem

4.1.4 Typewriter

Třída implementuje rozhraní *I_InitDraw*. Po správné inicializaci lze přímo vypisovat text. Při vykreslování zde není získána MVP transformace z třídy *Camera*. Tato transformace je při každém volání *updateText* znova sestavena a spočítána. Třída poskytuje navíc tyto funkce:

- `void updateText(int x, int y, string text)` – funkce, kterou je nezbytné zavolat před vykreslením textového řetězce. Předávané parametry jsou souřadnice výstupního zařízení, na kterých se má text v argumentu vykreslit.
- `void loadGlyphInfos()` – funkce, která při inicializaci načte veškeré informace o použitém fontu, a dále načte jeho bitmapou do paměti grafické karty.

Pro každý načtený symbol je naplněna struktura *Glyph*, která uchovává nezbytné informace k vykreslení. Jednotlivé struktury jsou uchovávány v datové struktuře *slovník*. V tomto slovníku jsou zaindexovány pomocí odpovídající ASCII hodnoty. Po každé aktualizaci textu musí být explicitně zavalána metoda `Draw()`. V opačném případě není text zobrazen. Metoda `Draw` potom předává vertex shaderu jednotkovou transformační matici MVP, protože pozice jednotlivých vrcholů už byla předem vypočítána v metodě `updateText()`.

4.1.5 UserInput

Ve třídě *ScreenManager* je vytvořena jediná instance této třídy. Třída poskytuje následující zpětná volání pro uživatelský vstup, která musí být na začátku programu navázána.

- `on_mouseEvent(int button, int state, int x, int y)` – tato funkce je volána kdykoliv je stlačené tlačítko nebo kolečko myši.
- `on_mouseMoveEvent(int x, int y)` – zpětné volání aktivované při pohybu myší.
- `on_specialKeyboard(int c, int x, int y)` – zpětné volání při stisku kláves se speciálním významem (ALT, SHIFT, CTRL).
- `on_keyboard(int key, int mouse_x, int mouse_y)` – zpětné volání při stisku běžných kláves klávesnice.
- `on_mouseMultipleMoveEvent(int c, int x, int y)` – ve verzi knihovny FreeGLUT pro OS Windows je tato funkce volána při stisku speciální klávesy a pohybu myší.

Třída dále rozlišuje dva stavy pro korekci kamery nebo mračna ve výčtovém typu *Ui_Camera_Mode*. Další výhodou této třídy je, že sjednocuje rozdílná zpětná volání knihovny FreeGLUT a také zpracovává rozdílné události, které se generují při aktivaci tlačítek myši. Při aplikování oprav na jednotlivá mračna si pamatuje předchozí stav kamery a naopak.

4.1.6 CommandBase

Jedná se o bázovou třídu, ze které jsou odvozeny všechny třídy, které poskytují funkcionality undo/redo. Třída má přetížený operátor pro porovnání `operator=()`. Třída dále definuje následující proměnné a metody:

- `string name` – veřejná proměnná, která je nastavena na odpovídající jméno v odvozených třídách.
- `virtual size_t getSize()` – slouží pro vracení skutečné velikosti paměti, kterou třída zabírá.
- `virtual void execute()` – po zavolání této funkce implementující třída provede požadovanou funkcionalitu.

Třídy, které poskytují veškerou funkcionalitu undo/redo programu a jsou od této třídy odvozeny, jsou:

- `UndoToggleVisibilityCommand` – poskytuje změnu viditelnosti mračna.
- `UndoChangePointSizeCommand` – vrátí předchozí velikost bodů.
- `UndoColorCommand` – poskytuje návrat předchozí barvy mračna.
- `UndoChangeCloudCommand` – poskytuje návrat změny mračna.
- `UndoSelectPointCommand` – smaže posledně vybraný bod z kolekce vybraných bodů.
- `UndoClearSelectedVerticesCommand` – při vykonání obnoví kolekci vybraných bodů do původního stavu.
- `UndoUpdateSceneCommand` – slouží pro návrat kamery do předchozí pozice a nastavení jejich předchozích parametrů. Veškeré potřebné hodnoty uchovává ve struktuře *Snapshot*.

Všechny výše uvedené třídy existují také ve variantě `Redo...` a poskytují přesně opačnou funkcionalitu.

4.1.7 InfoPanel

Třída informačního panelu kromě děděných metod implementuje tuto funkci:

- `toggleVisibility()` – přepíná vnitřní booleovskou proměnnou, podle které se rozhodne, jestli se má informační panel vykreslit.

Ve svém inicializačním cyklu vytváří instance tříd *ScreenOrigin* a *Logo*. Během vykreslování je nejprve v metodě `Draw()` změněna promítací transformace z perspektivní na rovnoběžnou. Dále je zde vypočítán počet snímků za vteřinu FPS a tato hodnota je spolu s ostatními informacemi vykreslena. Na konec vykreslovacího cyklu je vypočtena pozice pro vykreslení ostatní grafiky informačního panelu. Pro tuto grafiku je nastaveno odpovídající zvětšení.

4.1.8 ScreenOrigin

Tato třída má na starosti zobrazení orientace kamery v prostoru. Třída dědí od *I_InintDraw* a tak je možné ji inicializovat a vykreslit. Implementuje pouze jedinou veřejnou metodu a to:

- `setScale(float value)` – Po zavolání metody `Draw()` přidává k aktuální modelovací matici škálovací matici vytvořenou podle předaného parametru.

4.1.9 Logo

Třída se stará o vykreslování grafiky, která reprezentuje pohyb nebo změnu parametrů kamery. Ve své inicializační části načítá externí obrázky, které převádí na textury a ukládá je na grafickou kartu. Mezi těmito texturami následně ve vykreslovací části přepíná a index na vybranou texturu předává k dalšímu zpracování fragment shaderu.

4.1.10 PointCloudManager

Jedná se o manažer načtených mračen. Jeho hlavní funkcí je spravovat kolekci načtených mračen a udržovat si informaci o právě vybraném mračnu. Dále umožňuje vybraným mračnům měnit velikost bodů, barvu atd. Důležité metody pro tuto třídu jsou:

- `loadClouds(cmd_args_holder& cmd_args)` – vytvoří mračna z parametrů předaných v příkazovém řádku. Zpracuje konfigurační soubor s mračny a popř. nastaví správnou barvu.
- `void setDrawLowFPS(bool value)` – nastaví všem mračnům v kolekci omezený počet bodů pro jejich vykreslování.

4.1.11 BaseEntity

Jedná se o bázovou třídu, která kolem své vizuální reprezentace vykresluje drátěnou krychli. Současně je v nastaveném středu vykreslena lokální souřadnicová soustava, která slouží pro lepší reprezentaci objemu v prostoru. V odvozených třídách musí být přepočítány minima a maxima hodnot na všech osách a správně nastaven střed krychle. Pro zobrazení zvýraznění slouží chráněná proměnná *m_draw_wireframe*.

4.1.12 Pointdata

Tato třída slouží k reprezentaci prostorových informací o jednotlivých mračnech. Jako jediná obsahuje konstruktor, který jako parametr přijímá cestu k datovému souboru a při chybě během načítání vyvolává výjimku. Třída implementuje metody rozhraní *I_InitDraw* a zároveň dědí od třídy *BaseEntity*. Pro zpracování datového souboru s mračnem vnitřně využívá metod, které poskytuje třída *pcd_data*. Jedná se o konkrétní implementaci načítání a ukládání dat ve formátu *pcd*.

Důležité metody této třídy jsou zejména:

- `PointData(string path)` – konstruktor programu. V jeho těle se zpracovává datový soubor a dynamicky se alokují prostředky pro uložení vlastních dat, jako jsou vrcholy, popř. barva. Při volání konstruktoru třídy je nezbytné zpracovávat případné výjimky.
- `doCalculations()` – inicializační metoda, ve které dochází k výpočtu těžiště mračna a jeho extrémních hodnot v jednotlivých osách, a inicializaci paměťových objektů na grafické

kartě. Tato metoda je volána v metodě `Init()` před voláním metod bázové třídy. Jako střed pro vykreslení os je nastaven medián hodnot bodů mračna.

- `setTransformationParameters(...)` – pomocí této metody jsou instanci objektu předány korekční parametry, které jsou definovány v pomocném datovém souboru při načítání více mračen. Tyto parametry jsou při dalších výpočtech dále zpracovány.

- `getVertices()` – ze surové reprezentace všech vrcholů v paměti vytváří vektor vrcholů *glm::vec3*. Na tyto vrcholy jsou před vracením k dalšímu zpracování aplikovány lokální translace mračna a jeho rotace kolem lokálních os.

Třída dále rozlišuje mračna, která definují barvu pro jednotlivé vrcholy a která ne. Podle této informace dynamicky alokuje paměť pro barevnou složku dat. Paměť je potom dále předána ke zpracování grafické kartě. V případě, že není barevná složka definována, program definuje konstantní barvu, kterou předá ke zpracování grafickému shaderu.

4.1.13 Log

Tato třída obsahuje dvě statické metody pro vypisování informací během načítání a běhu programu. Obsahuje dvě základní statické metody.

- `void Log::WriteLog(log_level, string)` – vypíše na standardní výstup předaný řetězec ve formátu `log_level » text zprávy`.

- `void Log::WriteLogF(log_level, string, ...)` – stejné funkce jako předchozí metoda, která ale zároveň umožňuje předávat formátovací řetězec pro variabilní argumenty.

Pro odlišení několika úrovní důležitosti zpráv je k dispozici následující výčet `log_level` hodnot.

- `Log::debug` – slouží pro výpis informací, které se budou zobrazovat pouze tehdy, je-li program přeložen s definicí DEBUG.

- `Log::info` – slouží pro zobrazení standardních informací, jako například souřadnic vybraného bodu nebo vzdálenosti mezi body.

- `Log::warning` – zobrazuje základní varování programu. Jedná se o drobná varování jako jsou např. pokus změnit barvu, při kterém zároveň není vybráno žádné mračno.

- `Log::error` – chybový stav error naznačuje závažnou chybu v programu. Program nicméně dokáže v omezené funkcionalitě dále pracovat. Jako příklad lze uvést pokus o načtení neexistujícího mračna bodů.

- `Log::critical` – jedná se o nejvyšší kritický chybový stav, ve kterém program nedokáže dále správně pracovat. Tento stav může nastat např. při nemožnosti dynamicky alokovat další paměť pro novou barvu mračna nebo pokusu vytvořit nový záznam pro undo/redo

funkcionalitu. Program by měl po vypsání této chyby uvolnit všechny zdroje a korektně se ukončit.

Pokud nebyl program přeložen v režimu pro ladění, je možno si i tak nechat vypisovat ladicí logovací zprávy. Pro zobrazení je nutno program spustit s parametrem *-d*.

4.1.14 Configurator

Třída obsahuje několik statických metod pro získávání nastavených parametrů v konfiguračním souboru programu. Konfigurační soubor je detailně popsán v kap. 5.3. Jedná se o jedinou třídu, ve které jsou odděleně napsány funkce pro vytváření adresářů a získávání cesty ke konfiguračnímu souboru podle konkrétního operačního systému. Postup získání cesty ke konfiguračnímu souboru je znázorněn na obr. 8. Hlavní metody jsou:

- `int getline(char** buffer, size_t length, File* fd)` – metoda, která ze souboru načte kompletní řádek a výsledek uloží do předaného bufferu. Pokud nebyla alokována pro buffer dostatečně velká paměť, předchozí paměť je uvolněna a je alokována paměť pro nový buffer o dostatečné kapacitě.
- `char* getHomeDir()` – vrací cestu k domovské složce aktuálně přihlášeného uživatele.
- `char* getConfigFilePath()` – hledá konfigurační soubor v aktuálním adresáři, ze kterého byl program spuštěn. Pokud zde není soubor nalezen, prohledávání pokračuje k domovskému adresáři uživatele a adresáři *.ibisviewer*. Jestli ani v domovském adresáři není konfigurační soubor nalezen, je zde vytvořen nový, výchozí soubor se zakomentovanými řádky pro jednotlivé nastavitelné parametry.

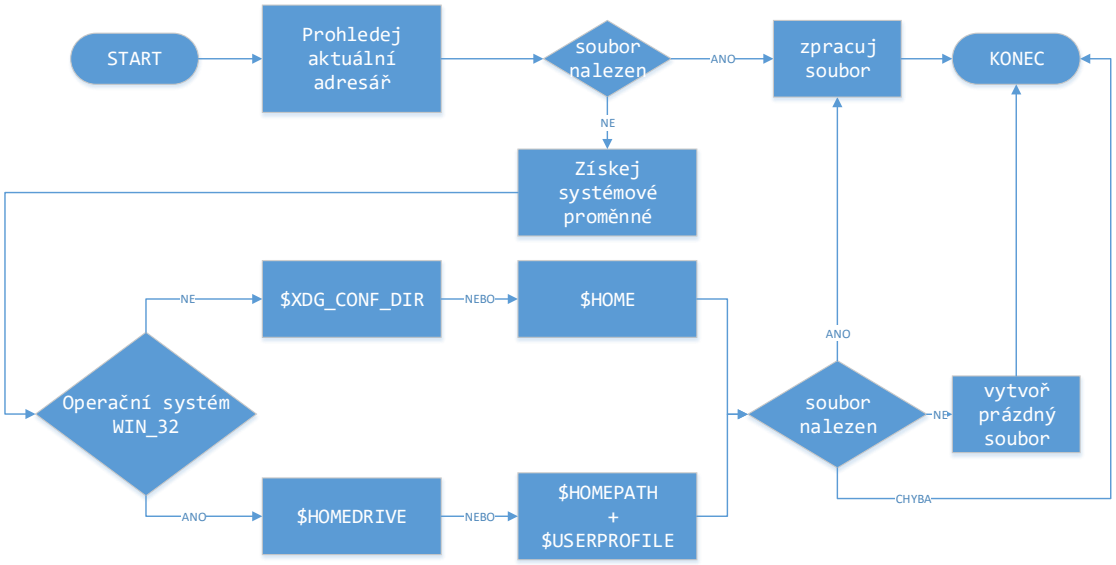
K načtení konfiguračního souboru dochází při prvním volání libovolné funkce k získání konfigurační hodnoty. Po prvotním načtení se již ke konfiguračnímu souboru dále nepřistupuje.

4.1.15 ColorRing

Tato třída slouží pro cyklické procházení předdefinovaných barev. Mezi její hlavní metody patří:

- `void addColor(Color)` – do fronty vloží další barvy.
- `Color nextColor()` – vrátí následující barvu z cyklické fronty barev.
- `Color prevColor()` – vrátí předchozí vracenou barvu z cyklické fronty barev.

Při cyklickém procházení je jako poslední barva vracena speciální barva *COLOR_RAMP*. Při zpracování této konstantní barvy je pro aktuální mračno vytvořena barevná škála a všechny body jsou obarveny podle jejich vzdálenosti na vlastní ose x. Barevná škála pro obarvení je vidět na obr. 9. Funkce pro samotné obarvení vrcholu je vidět ve výpise č. 1. Tento kód byl použit z prezentace k předmětu Vizualizace Dat [13].



Obrázek 8: Diagram získání cesty ke konfiguračnímu souboru

Obrázek 9: Barevná škála použita k obarvení bodů mračna

```
glm::vec3 ColorRamp( const float value, const float min_value, const float
    max_value, glm::vec3 * ramp, const int ramp_size ) {
if ( value >= max_value )
   return ramp[ramp_size - 1];
else {
   float a = ( value - min_value ) / ( ( max_value - min_value ) / ( ramp_size
       - 1 ) );
   const int band = _cast<int>( floor( a ) );
   a -= band;
   return ramp[band] * ( 1 - a ) + ramp[band + 1] * a;
} }
```

Výpis 1: Funkce pro obarvení mračna bodů pomocí barevné škály

4.1.16 ShaderUtils

Tato třída obsahuje tři statické metody, které jsou popsány níže.

- `char* getShaderSource(const char* shader_name)` – vrací zdrojový kód, který byl načten z externího souboru za běhu programu.
- `void printLog(GLuint object)` – v případě nezdaru při vytváření nebo získávání ukazatelů z programovatelných shaderů dokáže z OpenGL získat relevantní zprávu o poslední chybě.
- `GLuint createShader(const char* filename, GLenum shader_type)` – funkce, která zjednodušuje vytváření programovatelných shaderů. Kompiluje jednotlivé shadery a v případě nezdaru vypíše podrobnou informaci o chybě.

4.2 Programovatelné shadery

Moderní verze OpenGL ke své funkcionalitě využívají programovatelné shadery. Shadery reprezentují krátké podprogramy, které jsou vykonávány na grafické kartě. Program Ibis PCD Viewer pro svou funkci definuje právě dva shadery.

Prvním z nich je vertex shader, který je ve výpise č. 8 na str. 64. Každý shader musí začínat definicí verze GLSL, pod kterou bude kompilován. Program používá základní specifikaci GLSL, a proto si vystačí s její základní verzí. Na řádcích 3 - 5 jsou nadefinovány vstupní body shaderu a jejich datové typy. Tyto proměnné obsahují předané hodnoty pro zpracování programovatelným řetězcem.

Na řádcích 7 - 14 jsou uniformní proměnné, které jsou společné pro všechny shadery. Uniformní proměnné programu hrají roli konstant, které se během jednotlivých volání `glDraw()` nemění. Na řádcích 17 a 18 jsou nadefinovány výstupní proměnné vertex shaderu, které musí mít stejné jméno jako vstupní proměnné fragment shaderu.

Samotné vykonávání programu začíná v metodě `main()`, která také jako jediná musí být povinně definována.

Druhým nezbytným shaderem, který musel být vytvořen, je fragment shader. Jeho ukázka je zobrazena ve výpise č. 9 na str. 65. Má podobnou strukturu jako předchozí vertex shader. Veškeré vstupní proměnné musí nést stejné jméno a typ jako výstupní proměnné ve vertex shaderu.

Výstupem fragment shaderu je proměnná *color_out*, ve které je uložena barva pro vypočtený pixel výstupního zařízení. V metodě *main()* je na základě uniformních proměnných rozhodováno, jestli má být pro výstupní barvu použita uložená textura (řádek 22) nebo je shader použit pro vykreslení textu (řádek 16). Na řádku 43 lze vidět rozhodovací proces, který určí, jestli se má výstupní barva vypočítat z variabilní vstupní proměnné pro barvu, anebo se má použít konstantní barva.

```
info face="Lucida Console" size=16 bold=0 italic=0 charset="" unicode=1
   stretchH=100 smooth=1
aa=1 padding=0,0,0,0 spacing=1,1 outline=0
common lineHeight=16 base=13 scaleW=256 scaleH=256 pages=1 packed=0 alphaChnl=1
    redChnl=0 greenChnl=0 blueChnl=0
page id=0 file="font_lucida_0.bmp"
chars count=191
char id=32  x=161  y=75   width=3    height=1    xoffset=-1   yoffset=15
   xadvance=10   page=0 chnl=15
char id=33  x=250  y=14   width=3    height=12   xoffset=3    yoffset=1
   xadvance=10   page=0 chnl=15
                ⋮
```

Výpis 2: Výpis datového souboru s parametry pro jednotlivé grafémy

4.3 Vykreslování textu

Grafická nástavba OpenGL GLUT neposkytuje podporu pro vykreslování textu pomocí programovatelných shaderů. Pro každé vykreslení textového řetězce by bylo potřeba přepnout kontext OpenGL programu do *compability* profilu, kdy je využívám fixní vykreslovací řetězec, a potom zpět. Tento 'zastaralý' režim podle standardu ovšem nemusí být podporovaný na nových ovladačích.

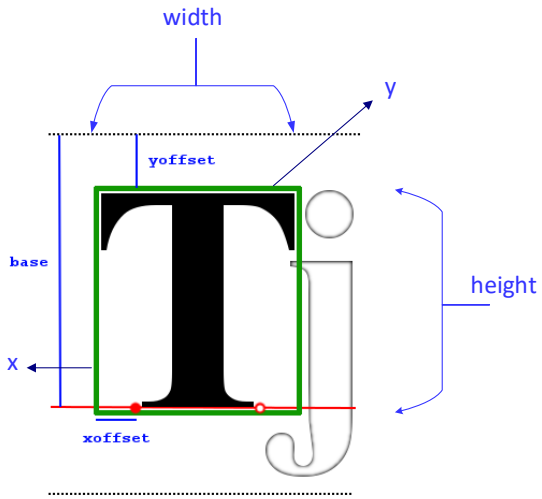
Z tohoto důvodu bylo třeba přistoupit k naprogramování vlastního řešení vykreslování textu pomocí programovatelného shaderu. Vzhledem k jednoduchosti řešení byla veškerá funcionalita zabudována do jediného fragment shaderu, který je společný pro celý program. Na obr. 12 je zobrazena bitmapa s fontem pro základní ASCII abecedu. Tato bitmapa byla vytvořena programem CBFG [12].

Součástí výstupu programu pro tvorbu fontů je i datový soubor ve formátu *fnt*. Jedná se o jednoduchý datový formát, jehož ukázka je ve výpise č. 2.

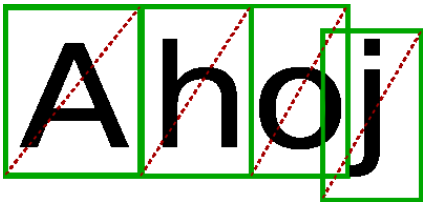
Tento datový soubor je zpracovaný třídou *Typewriter*. Pro každý znak k vykreslení je nutné vypočítat jeho pozici na zobrazovacím zařízení, popř. jeho orientaci vůči pozici kamery. Při zpracování nejsou použity všechny parametry. Program z hlavičky zpracovává pouze informaci o jménu souboru s bitmapou *file*. Každé písmeno je v souboru uvozeno značkou *char* a jeho *id* odpovídá zakódování znaku v ASCII abecedě. Význam dalších parametrů je znázorněn na obr. 10. Tyto parametry slouží ke správné lokalizaci jednotlivých písmen v bitmapě, popř. k výpočtu korekčních parametrů vzhledem k výchozí poloze a základní čáře (hodnota *base*). Písmena jsou potom vykreslena blíže k sobě. Písmena, která zasahují pod základní čáru (g,y atd.), musí být správně vertikálně posunuta.

Základní požadavky na vykreslovaný text byly:

- Normála každého písmena textu musí směřovat do centra kamery.



Obrázek 10: Zobrazení parametrů pro jednotlivé grafémy



Obrázek 11: Skládání jednotlivých písmen do celých řetězců

- Všechna písmena musí být všude stejně velká – tzn. nezávisle na vzdálenosti od kamery.

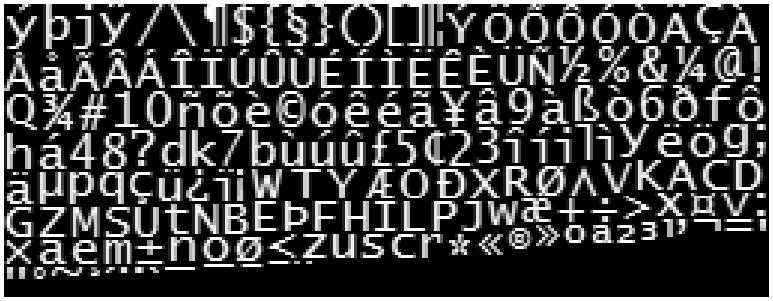
Pro správné umístění trojúhelníků, ze kterých se písmena skládají, bylo třeba využít jednu z metod billboardingu popsanou na internetových stránkách [5]. Jedná se o základní metodu, která splňuje výše popsané požadavky. Na výpise 3 lze vidět celý postup výpočtu. Standardním způsobem se vypočítá počáteční umístění na monitoru pomocí maticového násobení $Projection \times View \times Model$ a následně se provede normalizace. Pro všechny vrcholy, které tvoří jednotlivá písmena textu, jsou dopočítány offsety v závislosti na nastaveném rozměru zobrazovacího zařízení. Aktuální hodnota šířky a výšky zařízení se zjišťuje OpenGL voláním a musí být správně nastavena před vykreslováním písma. Kromě správného umístění jsou shaderu předány informace o korespondujících texelech. Texely jsou souřadnice, které odpovídají souřadnicím v uložené textuře. Proměnná horizontální offset je po jednotlivých písmenech inkrementována o jejich šířku tak, aby další písmeno začínalo vedle předchozího.

Tento zbytek kódu zde není z důvodu své délky a relativní složitosti uveden. Při výpočtu umístění dlouhých textů není počítáno se zalamováním. Text přesahující hranice zobrazovacího zařízení je oříznut.

```
glm::mat4 pm = ScreenManager::GetManager()->m_perspective_matrix;
glm::mat4 view_matrix = ScreenManager::s_camera->getViewMatrix();
glm::vec4 billboard_position(m_x_pos, m_y_pos, m_z_pos, 1.0f);      //create
    model position
glm::vec4 billboard_pos_camera_space = pm * view_matrix * billboard_position;
billboard_pos_camera_space /= billboard_pos_camera_space.w;

float view_m[4];
glGetFloatv(GL_VIEWPORT, view_m);
float screen_width = view_m[2];
float screen_height = view_m[3];
float h_offset = 0;
```

Výpis 3: Výpočet pozice vykreslovaného textu na obrazovce



Obrázek 12: Bitmapa s použitým fontem

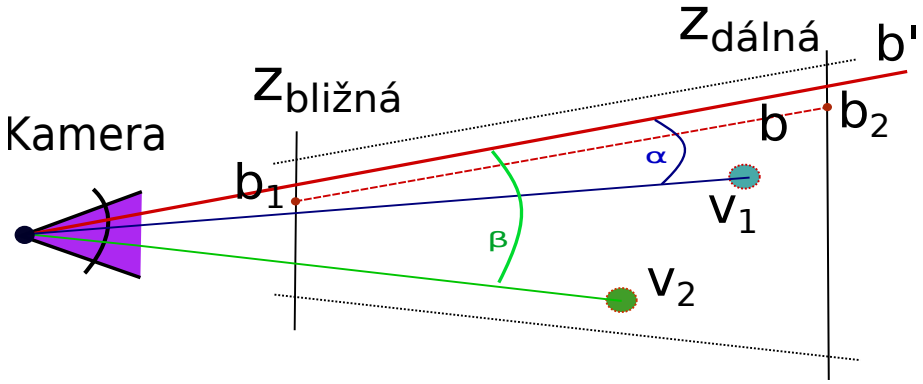
Jednotlivá písmena textového řetězce jsou vytvořena pomocí dvou základních primitiv OpenGL – trojúhelníků. Náznak takto vygenerovaného textu je na obr. 11. Na každý trojúhelník je potom ve fragment shaderu nanesena správná část textury pro konkrétní písmeno.

4.4 Výběr a vykreslování vybraných bodů

Všechny vybrané body jsou vykreslovány až po vykreslení mračen bodů a zároveň před vykreslením informačního panelu. Uloženy jsou v proměnné *m_selected_points* ve třídě *ScreenManager*. Kvůli zvýraznění je nastavené trojnásobné zvětšení oproti velikosti bodů zobrazených mračen. Mezi každými dvěma body je vykreslena úsečka.

Výběr bodů probíhá za pomocí funkce `gluUnProject()`. Tato funkce převádí předané souřadnice kliknutí myší na obrazovku na souřadnice OpenGL soustavy. Kromě souřadnic kliknutí myší se funkci musí předat ještě souřadnice hloubky z.

Řešením bylo vypočítat souřadnice bodů dotyku v základní souřadné soustavě OpenGL b_1, b_2 na přední a zadní ploše zobrazovacího objemu. Postup řešení je naznačen v obr. 13. Z tohoto



Obrázek 13: Výběr bodu za pomocí měření úhlů

důvodu je funkce pro inverzi transformačních matic volána dvakrát. Poprvé s parametrem $z = 0.05$ (vzdálenost k bližší rovině) a dále s parametrem $z = 1.0$ (zobrazovací rovina v nekonečnu).

Takto získaný vektor b je posunut do počátku soustavy $o = (0, 0, 0)$. Posunem vznikne vektor b'. Vektory k porovnání vzniknou mezi počátkem soustavy o a souřadnicemi testovaných bodů. Všechny vektory jsou před dalším zpracováním normalizovány. Mezi těmito vektory je vypočítán kosinus úhlu (pomocí skalárního součinu). Úhel je následně porovnán s nejmenší doposud zjištěnou hodnotou kosinu úhlu, dokud není zjištěna minimální hodnota úhlu pro všechny body. Pokud bod s nejmenší hodnotou úhlu není v kolekci vybraných bodů, a zároveň je svíraný úhel menší než cca $1,8°$, je do této kolekce přidán.

5 Popis použitých datových souborů

Program ke své činnosti musí zpracovávat několik datových souborů. V první řadě se jedná o soubory, ve kterých jsou uloženy prostorové informace o naměřených bodech. Dále se jedná o různé konfigurační soubory, které definují parametry spuštění samotného programu a jeho následné chování při vykreslování bodů. V následující části najdeme přehled těchto datových souborů.

5.1 Point Cloud Data *pcd*

Program je schopný načítat textové soubory s prostorovými daty ve formátu point cloud data. Jedná se o formát popsaný týmem, který vyvíjejí knihovnu PCL[4] pro práci s prostorovými daty. Knihovna je součástí otevřeného projektu, který se zabývá zpracováním mračen bodů.

Data z *pcd* souboru je program schopný načítat jak v textovém ASCI formátu, tak v binární podobě. Ve výpise č. 4 je ukázka souboru s 10 body.

```
1  VERSION .5
2  FIELDS x y z~rgb
3  SIZE 4 4 4 4
4  TYPE F F F F
5  COUNT 1 1 1 1
6  WIDTH 10
7  HEIGHT 1
8  POINTS 10
9  DATA ascii
10 0.000000 1.000000 20.000000 4.2108e+06
11 0.000000 2.000000 19.969559 4.2108e+06
12 0.000000 1.000000 19.878281 4.2108e+06
13 0.000000 2.000000 19.726181 4.2108e+06
14 0.000000 4.000000 19.513306 4.2108e+06
15 0.000000 3.000000 19.239700 4.2108e+06
16 0.000000 3.000000 19.905487 4.2108e+06
17 0.000000 2.000000 19.510742 4.2108e+06
18 0.000000 1.000000 19.055588 4.2108e+06
19 0.000000 0.000000 19.540161 4.2108e+06
```

Výpis 4: Datový soubor v *pcd* formátu

Na řádku 1 najdeme číslo verze formátu, v tomto případě 0.5. Na řádku 2 vidíme formát uložených dat. Pokud soubor obsahuje pouze souřadnice jednotlivých bodů, najdeme zde `x y z`. Obsahují-li data i informaci o barvě, najdeme zde `x y z rgb`. Na řádku 3 je definovaná velikost jednotlivých složek v bytech. Na řádku 4 je definovaný jejich datový typ, který byl použit

pro uložení. Na řádku 5 je uložený počet elementů v jednotlivých dimenzích. Program dokáže pracovat pouze se složkami velikosti 1. Řádky 6 a 7 popisují počet bodů na výšku a šířku. Standardně je jako výška uloženo číslo 1, ale například při stereoskopickém snímání bychom mohli mít v jednom souboru uloženy dvě sady bodů. Řádek 8 udává celkový počet bodů. Tato hodnota se musí rovnat $výška \times šířka$. Řádek 9 signalizuje, zdali jsou následují data zapsána v ASCII (textovém) nebo binary (binárním) formátu. Zápis v binárním formátu má výhodu v rychlosti načítání resp. ukládání do souboru a v celkově menší velikosti datového souboru.

Ostatní řádky již obsahují samotné souřadnice.V tomto případě řádky obsahují i informaci o barvě jednotlivých bodů. Speciálně na řádku 10 je definován bod o souřadnicích $x = 0$, $y = 1$, $z = 20$ a dále je zde informace o barvě, která je v *pcd* souborech uložena jako datový typ float.

```
float clri = readColor(row);
int r = ((int)clri >> 16) & 0x0000ff;
int g = ((int)clri >> 8) & 0x0000ff;
int b = ((int)clri)     & 0x0000ff;
```

Výpis 5: Rozkodování barvy ve formátu *pcd*

5.2 Konfigurační soubor pro načítání více mračen

K načtení více než jednoho datového souboru s mračnem bodů je potřeba použít konfigurační soubor. V každém řádku souboru je cesta k datovému souboru s mračnem ve formátu *pcd*. Dále je zde možnost definovat barvu ve formátu $\#RRGGBB$, kde jednotlivé složky barev jsou jednobytové hodnoty barvy v hexadecimálním zápisu. Jako poslední následují transformační parametry, jako jsou rotace kolem jednotlivých os a vzdálenost od středu na jednotlivých osách. Řádky, které začínají znakem # jsou při zpracování vynechány. Tento znak lze zároveň použít jako komentář v konfiguračním souboru. Ve výpise č. 6 je ukázka souboru s několika nadefinovanými mračny.

```
1 pcd/kancl.pcd
2 pcd/stat1-001.pcd #ff0000
3 pcd/stat1-004.pcd #0000ff -2470 0.0 0.0 0.0 -10.23 0.0
4 #pcd/stat1-008.pcd #880060 1490 0.0 -140 0 0 0
5 #pcd/stat1-010.pcd #ff8870 1800 40.0 3850 0 0 0.0
```

Výpis 6: Konfigurační spouštěcí soubor

Na výpise výše můžeme vidět jednoduchý konfigurační soubor. Na řádku 1 je pouze cesta k souboru s mračnem. Ve druhém řádku lze vidět definice červené barvy pro načítané mračno. Dále je na řádku 3 definována modrá barva pro načtené mračno. V tomto řádku je zároveň definovaný posun o 2470 jednotek na ose $-x$ a rotace kolem osy y o 10.23 stupňů (proti směru hodinových ručiček). Řádek číslo 4 a 5 se nezpracovává.

5.3 Konfigurační soubor programu

Při prvním spuštění programu se v domovském adresáři aktuálního uživatele vytvoří složka *.ibisviewer* a v ní konfigurační soubor *ibis_config*. Tento konfigurační soubor slouží k nastavení úplné cesty k programu a několika dalších parametrů pro definování způsobu vykreslení scény. Ve výpise č. 7 je ukázka konfiguračního souboru.

```
1 #APP_PATH==c:\\users\lukas\projects\ibis pcd viewer\debug\
2 #POINT_SIZE==5
3 USE_VERTEX_SHADER_POINT_SIZE==1
4 USE_POINT_PARAMETERS==1
5 #USE_SMOOTH_POINT_SPRITES==0
6 #CAMERA_DISTANCE==10
```

Výpis 7: Konfigurační soubor programu

Každý řádek, který začíná znakem #, se nezpracovává. Parametr *APP_PATH* určuje cestu, která bude použita pro načítání pomocných dat programu jako jsou shadery, fonty nebo obrazové bitmapy v případě, že se program spouští z jiného adresáře. POINT_SIZE definuje výchozí velikost vykreslovaného bodu v pixelech. Parametr USE_VERTEX_SHADER_POINT_SIZE byl vytvořen z důvodu nemožnosti nastavit velikost vykreslovaného bodu ve vertex shaderu u některých ovladačů grafických karet ATI. V případě že se ve scéně nevykreslují žádné body, doporučuji nastavit tento parametr na 0. Na řádku 4 lze potlačit funkcionalitu změny velikosti bodu v závislosti na vzdálenosti od kamery. Parametr USE_SMOOTH_POINT_SPRITES slouží k vykreslování kulatých bodů. Při deaktivaci jsou body vykreslovány jako čtverce. Zde je nutno podotknout, že tato funkcionalita musí být podporována grafickou kartou. Na posledním řádku lze nastavit vzdálenost kamery od středu otáčení. Výchozí hodnota je 10.

6 Popis uživatelské části

Program Ibis PCD Viewer je napsán tak, aby umožňoval interaktivní procházení virtuálního světa s velkým počtem bodů ve 3D prostoru. Ve scéně je zakomponována jedna kamera, u které uživatel může měnit množství parametrů jako je pozice a orientace v souřadné soustavě. Program po spuštění z příkazového řádku načte a zpracuje jednotlivé soubory s mračny bodů. Poté celou scénu přiměřeně zvětší, resp. zmenší. Nedochází zde k uniformnímu zvětšení jednotlivých mračen do jednotkových krychlí.

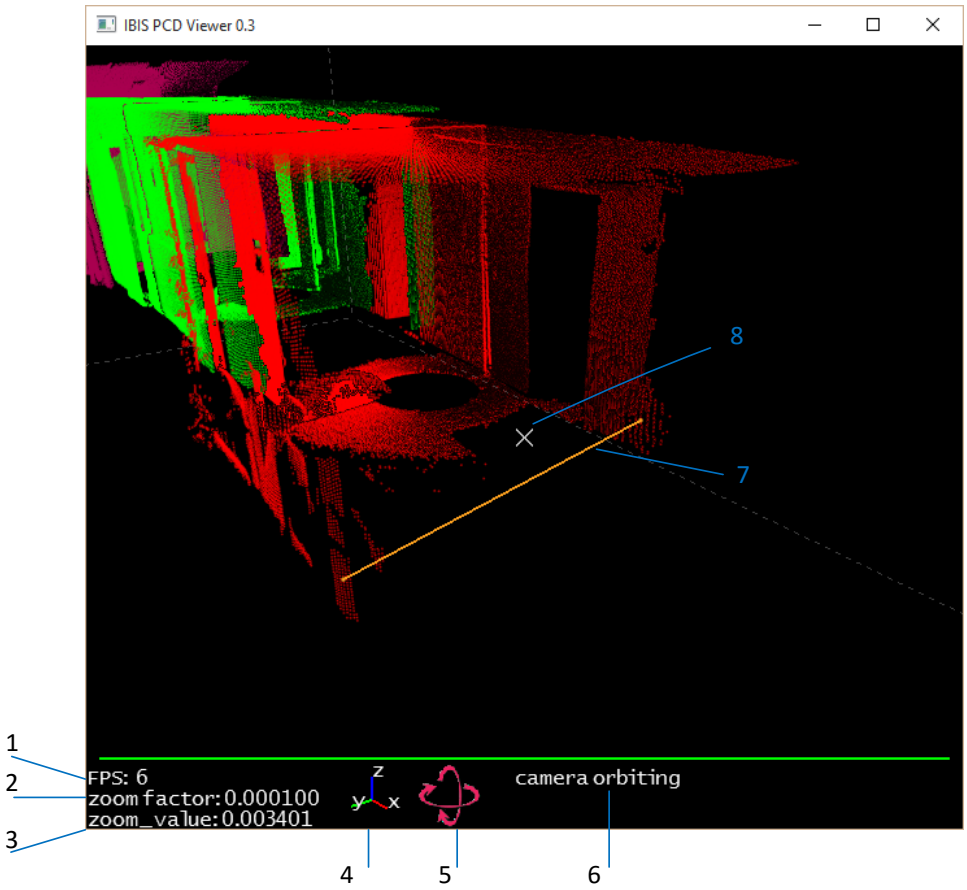
Na obr. 14 lze vidět ukázku programu s načtenými třemi mračny bodů. Hlavní plocha okna je věnována zobrazeným bodům jednotlivých mračen, která od sebe mohou být barevně odlišena. Ve spodní části okna je zelenou úsečkou oddělen informační panel, který může být podle potřeby schován. Další prvky jsou:

1. Zobrazení snímků za sekundu – v případě nízké hodnoty lze program spustit s parametrem '-c', a tím omezit počet vykreslovaných bodů mezi stisknutím, resp. uvolněním tlačítka myši.
2. Faktor škálování – slouží pro změnu jemnosti krokování přibližování kamery.
3. Hodnota přiblížení – jedná se o hodnotu, o kterou bylo mračno zmenšeno.
4. Pomocný kříž – zobrazuje natočení souřadné soustavy kamery.
5. Jednotlivé ikony – symbolizují zvolený režim změny parametrů kamery; jedná se např. o rotaci kolem středu, posun v horizontu atd.
6. Slovní popis aktuálně prováděného úkonu s kamerou nebo vybraným bodem – slouží k zobrazování krátkých zpráv uživateli.
7. Úsečka, která se vykreslí mezi dvěma vybranými body – při výběru dalších bodů v mračnu je dále prodlužována.
8. Značka, která symbolizuje střed otáčení kamery – kamera kolem tohoto středu rotuje nebo se současně s tímto bodem posouvá; uživatel má možnost kameru od středu otáčení libovolně oddálit nebo se přiblížit, a získat tak pohled 'první osoby'.

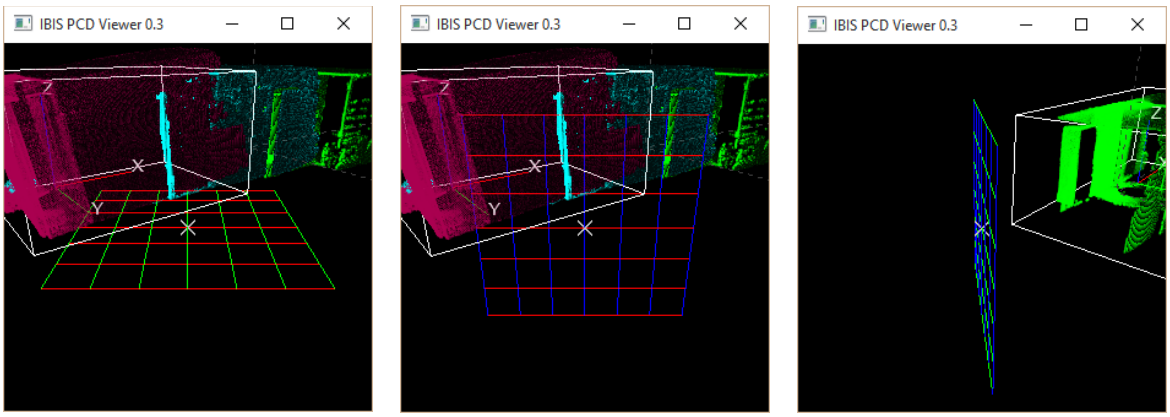
Při pohybu v jednotlivých osách nebo plochách se zobrazují čáry v barvě os, ve kterých se kamera posouvá. Tato zobrazení jsou znázorněna na obr. 15. Na snímcích je znázorněn rámeček kolem vybraného mračna. Pouze u takto označeného mračna lze interaktivně měnit jeho pozici a orientaci, stejně jako jeho barvu.

6.1 Hardwarové požadavky

Aplikace je nezávislá na použitém hardware. Pro spuštění je ovšem nezbytné mít nainstalovány ovladače grafické karty podporující OpenGL. OpenGL ve verzi ES není podporován. S touto



Obrázek 14: Hlavní okno programu



Obrázek 15: Posun v jednotlivých rovinách

verzi se lze nejčastěji setkat na mobilních zařízeních s OS Android nebo Raspberry Pi. Pokud počítač nedisponuje 3D grafickým akcelerátorem s potřebnými ovladači OpenGL, potom můžeme použít softwarovou emulaci OpenGL Mesa[9]. Samotný program byl testován na počítačových platformách s procesory x86 dále popsaných v kap. 8. Potřebná paměť je potom přímo závislá na množství načítaných mračen a množství bodů v nich uložených. Minimální množství potřebné paměti lze nalézt v tab. 5.

Program lze spustit na low-end zařízeních v režimu kompatibility s omezeným vykreslováním a počtem zobrazovaných bodů v rámci tisíců. Minimální konfigurace, na které byla otestována funkčnost programu, byla Intel Atom N270, 1 GB RAM, Intel Graphics Media Accelerator 945GSE Calistoga, OS Fedora 23, ovladač grafické karty Mesa 2.1.

6.2 Softwarové požadavky

Pro správnou funkcionalitu je nezbytné, aby použitý OS disponoval knihovnami OpenGL, FreeGLUT a GLEW. Tyto knihovny jsou více popsány v kap. 2. Knihovny OpenGL nejsou součástí instalace programu a je na uživateli, aby zabezpečil jejich aktuální instalaci.

6.3 Příprava a překlad programu

Program je distribuovaný na přiloženém DVD. Jeho součástí není žádný instalátor, který by zabezpečil vytvoření odkazů ke spuštění na ploše nebo zkopíroval všechny soubory nezbytné pro chod programu. Pro jeho otestování doporučuji zkopírovat kompletní DVD do libovolné složky na disku počítače. Během překladu jsou totiž kopírovány zdrojové obrázkové soubory nebo fonty, které jsou nezbytné pro správný chod programu. Kvůli zamezení duplicity jsou tyto soubory odkazovány v projektových souborech pomocí relativních cest.

Binární verzi programu lze na platformách s procesory x86 přímo spustit ze složky *projects/visual studio/Release/* v případě MS Windows nebo *projects/make/* v případě OS Linux. Pro OSX není k dispozici ani spustitelný soubor, ani připravený soubor makefile.

Pro spuštění v OS Windows je ještě nezbytné, aby OS disponoval dynamickými verzemi použitých knihoven (jedná se o soubory glut.dll a glew.dll, které by měly být v systémové složce pro sdílené knihovny). Pokud tyto soubory v OS chybí, je nutné je ručně nakopírovat z přiloženého DVD z adresáře s knihovnami *libs/*.

6.3.1 Překlad pod OS Windows

Pro správný překlad pod OS Windows je nezbytné v projektu nastavit cesty ke statickým knihovnám freeglut.lib a glew.lib, které obsahují informace pro použitý linker. Obě knihovny se nachází na přiloženém DVD ve složce *libs/*. Jedná se pouze o jejich 32bitové verze. Zájemce o 64bitové verze lze odkázat na stránky výrobce, kde se dají stáhnout také jejich aktuální verze s opravenými chybami.

Celý projekt pro MS Visual Studio 2013 lze nalézt na DVD ve složce *projects/visual studio*. V projektu jsou již přednastaveny cesty k linkovaným knihovnám, které jsou popsány výše. Součástí překladu je i zkopírování používaných dynamických knihoven do výstupního adresáře tak, aby bylo možné program přímo přeložit a spustit bez potřeby cokoli ručně kopírovat.

V projektu jsou také nastaveny parametry pro příkazový řádek, které obsahují relativní cestu k datovému souboru s mračnem bodů `chodba.pcd`. Pokud je projekt spuštěn z Visual Studia pomocí tlačítek Debug nebo Run, měl by se bez problémů přeložit a následně spustit a zobrazit okno s testovacím mračnem bodů.

6.3.2 Překlad pod OS Linux

Pro překlad na OS Linux je nezbytné mít nainstalovány knihovny uvedené v kap. 2. Pro uživatele Debianu lze doporučit balíky freeglut3, freeglut3.dev, glew-utils, libglew-deb a libglew1.7, které jsou běžně dostupné z repozitáře systému. Na OS Fedora je nezbytné nainstalovat balíky glew-devel a freeglut-devel, které jsou taktéž součástí repozitáře pro distribuci. OpenGL by mělo být dostupné po správné instalaci ovladačů pro konkrétní grafickou kartu, tzn. mělo by být dostupné ihned po spuštění X Window systému.

Pro softwarovou emulaci je možno využít knihovny mesa-common-dev, resp. mesa-libGL. Popis instalace jednotlivých balíčků s knihovnami je nad rámec tohoto textu a je součástí návodu k použití pro konkrétní verzi OS.

Překlad se na linuxových distribucích nejjednodušeji provádí pomocí nástroje GNU Make příkazem *make release*. Nezbytný soubor Makefile lze nalézt na přiloženém DVD ve složce *projects/make/*.

6.3.3 Spuštění

Samotné spuštění programu Ibis PCD Viewer se provádí z příkazového řádku terminálového emulátoru ve formátu:
`iviewer [parametry] mračno.pcd [-#barva_RRGGBB] | @soubory.pcd` .
Pro výpis akceptovaných parametrů a jednoduchou nápovědu lze spustit program s parametrem `iviever -h` nebo si vestavěnou nápovědu nechat vypsat do konzole stiskem klávesy *'?'*. Ukázka výpisu nápovědy je na obr. 21 na str. 67.

Přehled všech akceptovaných parametrů lze nalézt v tab. 4. Pro zobrazení více mračen současně slouží pomocný konfigurační soubor zobrazený ve výpise č. 6. Tento konfigurační soubor se musí v příkazovém řádku uvést s prefixem `'@'`. Ukázka miliónů načtených bodů lze vidět na str. 22.

Další konfigurační soubor, který se zpracovává při spuštění programu, je popsán v kapitole 5.3. Hlavním důvodem jeho zavedení byla potřeba spouštět program z jiného adresáře, než je samotný program. Při správném nastavení absolutní cesty k programu je umožněno správné načítání fontů a dalších pomocných souborů.

Parametr	Popis funkce
–version / -v	Zobrazí aktuální verzi programu iviewer.
-fo	Program se spustí v legacy módu a bude využívat fixní vykreslovací řetězec OpenGL.
–help / -h	Vypíše ovládání programu a možnosti jeho spuštění.
-p	Vypíše OpenGL informace o grafické kartě, jejich ovladačích a rozšiřujících funkcích.
-d	Pokud byl program zkompilován v Release verzi, do konzole se budou vypisovat ladicí informace.
-c	Během pohybu kamery se budou mračna vykreslovat s omezeným počtem bodů.

Tabulka 4: Parametry programu předávané při spuštění

6.4 Ovládání

Ibis PCD Viewer se ovládá pomocí myši a klávesnice. Hlavní důraz byl kladen na co nejmenší potřebu přesunu rukou po klávesnici nebo využití kombinace více kláves. Tabulka 6 na str. 69,popisuje veškeré ovládání a funkcionalitu programu.

Kromě široké možnosti nastavení kamery do požadované pozice a úhlu bylo během vývoje programu potřeba zavést drobné korekce pro jednotlivá mračna. Funkcionalitu jako je změna barvy mračna, přepnutí viditelnosti nebo zavedení korekce je možné aplikovat pouze na aktuálně vybrané mračno bodů. Vybrané mračno je zvýrazněno pomocí vizuální drátěné krychle, která je zobrazena kolem všech jeho bodů.

7 Dostupné prohlížeče pro zobrazování prostorových dat

V dnešní době existuje široké spektrum placených nebo volně dostupných prohlížečů prostorových dat, které lze použít na všech nejčastěji používaných operačních systémech. Většina dostupných programů pro prohlížení prostorových dat primárně pracuje s daty, která jsou získána z lidaru a uložena v jemu odpovídajícím formátu *las*, *laz*. Data bývají před zobrazením zpracována nebo filtrována a poté jsou využity moderní techniky vizualizace dat.

Většina těchto programů dokáže zobrazovaná data klasifikovat na vodstva, zástavby, porosty apod. Kromě zobrazení bodů v základních barvách dokáže vypočítat elevační mapy a vhodně jednotlivé body obarvit. Další standardní funkcí je vizualizace dat v kombinaci s uloženou texturou. Jednotlivé body jsou potom obarveny podle barevného podkladu, a více tak odrážejí zachycenou realitu.

7.1 Programy pro bezplatné použití

V této kapitole následuje krátký přehled dostupných programů, které jsou šířeny zdarma pro nekomerční využití a jsou k dispozici pro více než jeden OS. Většinou se jedná o studentské projekty, které byly dále rozvíjeny a vylepšovány malým týmem lidí.

PointCloudViz

Prohlížeč lidarových dat, který dále podporuje formáty *xyz, pts*. Jedná se o základní textové formáty uložení dat. Na každém řádku je x, y a z souřadnice jednotlivých bodů. Program disponuje bohatým GUI a dále např. načítáním dat ze vzdálených serverů. Tuto funkcionalitu se bohužel nepodařilo ověřit, protože program bez dalších informací havaroval.

Program poskytuje základní funkce jako výpočet elevačního modelu pro další zpracování v geoinformatice, tak klasifikaci dat, základní RGB zobrazení. Dále poskytuje plynulé načítaní bodů v závislosti na vzdálenosti kamery a výpočet barvy bodů na základě podkladových obrázků. PointCloudViz také poskytuje informace o vybraných bodech a výpočet elevace a dálky.

pcd_viewer

Jednoduchý prohlížeč, který je součástí instalace multiplatformní knihovny point cloud library. Program nedisponuje GUI, ale ovládá se pomocí myši a klávesových zkratek. Veškeré informace vypisuje do konzoly. Po načtení mračna ho nedokáže rozumně zmenšit do zobrazovacího objemu a uživatel tak po každém spuštění musí mračno zmenšit. Do některých pozic se při pohybu kamerou prostorem dá dostat pouze velice obtížně. Při absenci zpětné vazby o natočení kamery se do některých pozic dokonce nedá dostat vůbec. Veškeré parametry kamery nebo okna se dají nastavit z příkazového řádku při spuštění programu. Prohlížeč také dokáže načítat více mračen, která jsou definována na příkazovém řádku. Podporované formáty jsou *pcd, vtk*. Program by měl také podporovat výběr bodů, ale tuto funkci se pod OS Windows nepodařilo ověřit.

lidarview

Jedná se online webový prohlížeč prostorových dat ve formátech *las, xyz*. Zobrazuje hlavní okno, ve kterém se zobrazuje mračno bodů. V tomto okně lze volně ovládat pohyb kamery prostorem. V dalších oknech je mračno zobrazeno půdorysným a bokorysným pohledem. U kamery nelze měnit skoro žádné parametry, dokonce ani měnit hodnotu přiblížení v rozumných krocích. Zobrazené mračno je vykresleno pomocí hodnot klasifikace, elevace a intenzity bodů nebo v RGB barvách.

potree

Jedná se o další online projekt ke zobrazení lidarových dat. Pro svou funkci využívá WebGL. Podporuje nastavení širokého spektra parametrů, jako pozorovací úhel kamery, velikost a typ zobrazených bodů. Dále dokáže měřit vzdálenosti, úhly a objemy. Naměřené hodnoty zobrazuje přímo do vykreslovaného mračna. Při přiblížení kamery zjemňuje rozlišení zobrazovaných dat a dynamicky načítá potřebná data. Program je šířen zdarma jako open source ve formě zdrojových kódů.

plas.io

Taktéž se jedná o online projekt. Poskytuje podobnou funkcionalitu jako projekt *potree*. Pro vykreslení využívá WebGL. Zobrazované mračno dokáže obarvit podle výškové mapy nebo uživatelem nadefinované barevné mapy. U programu nelze měnit typ vykreslení bodu nebo nastavit výpočet velikosti bodu v závislosti na vzdálenosti od kamery. Podporované formáty jsou *las, laz*. Program dokáže v mračnu měřit vzdálenosti nebo interpolovat výslednou barvu podle uložené barvy a dalších klasifikací bodů. Nastavování parametrů probíhá pohodlně pomocí posuvníku v boční části okna. Program vykresluje všechny body najednou, a tak je při vysokém počtu bodů navigace v prostoru pomalejší. Zajímavá je funkce změny měřítka výšky.

Point Cloud Free Viewer

Program dovoluje vykreslit více než 10 milionů bodů v RGB barvách. K vykreslení primárně využívá grafický engine Unity společně s DirectX 11. Program podporuje poměrně velké spektrum formátů uložení - *xyz, xyzrgb, cgo, asc, catia asc, ply (asc), las, pts.*

CloudCompare

Jedná se o 3D zobrazovací program pro desktopové počítače, který dále dokáže data zpracovávat. Program byl vyvíjen pro zobrazování více než 100 milionů bodů. Na body dokáže aplikovat pokročilé techniky jako je registrace, změna hustoty bodů, změnu barev, uložených normál, výpočet statistických dat nebo automatická segmentace. CloudCompare obsahuje bohaté GUI,

které je vytvořeno v QT a pro vykreslení používá OpenGL. Program dokáže interaktivně jednotlivá mračna posouvat a rotovat. Primárním účelem tohoto programu bylo rozeznávat změny v naskenovaných datech.

7.2 Porovnání s Ibis PCD Viewer

Program Ibis PCD Viewer si klade za cíl široké nastavení parametrů kamery za běhu programu. Za pomoci konfiguračních souborů lze nastavit další parametry pro vykreslení. V další řadě poskytuje pohodlné zavádění korekcí pro již načtená mračna. Tuto funkci některé z ostatních prohlížečů nenabízejí. Dokonce u nich nelze načítat více souborů najednou. Pro tuto funkci je nezbytné datové soubory nejprve spojit za pomoci externího programu.

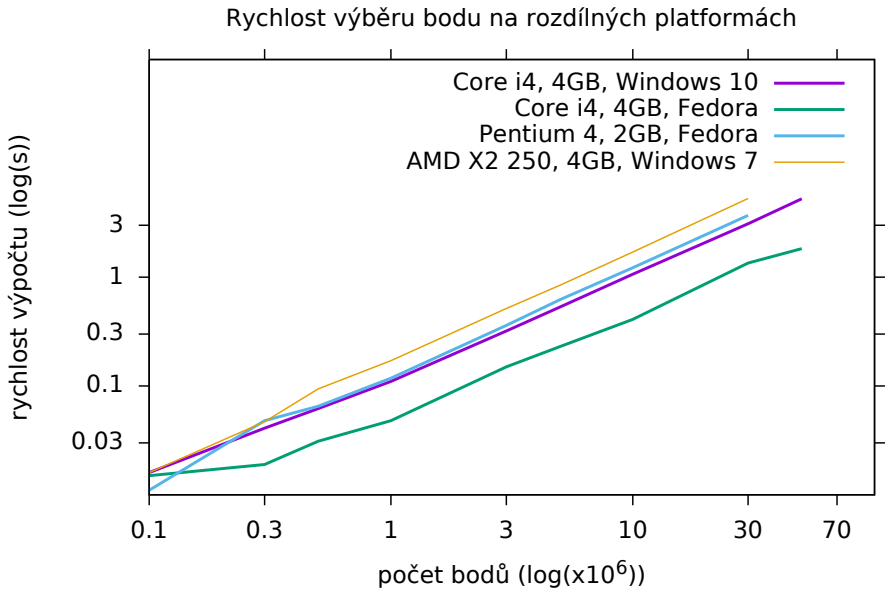
Program Ibis PCD Viewer se v základní funkcionalitě inspiroval programem *pcd_viewer*, zejména jeho absencí jakéhokoli GUI a pohodlného ovládání. Při nastavování parametrů kamery jde však dále a umožňuje mnohem více možností pohybů prostorem. Pokud je program spuštěn v režimu omezeného vykreslování bodů během pohybu kamery, patří v rychlosti vykreslování mezi rychlejší prohlížeče.

8 Výkonnost programu

Samotný program pcd viewer byl vytvářen s ohledem na využívání minimálního množství cizího kódu. Díky tomu bylo možné zvolit veškerou vnitřní reprezentaci dat a zamezit nadměrnému duplikování paměti. Vzhledem k předpokladu nasazení na moderních procesorech s dostatkem paměti RAM nebylo nutné přistoupit k ukládání dat v komprimovaných formátech, které jsou podporované standardem OpenGL. Nemalou část samotného spustitelného programu zabírá staticky sestavovaná knihovna pro matematické výpočty GLM. Tato knihovna je ale nezbytná pro základní funkci programu.

8.1 Rychlost výpočtu výběru bodu

Jelikož je tento výpočet prováděn na CPU, může pro velká mračna razantně zpomalit běh programu. Z tohoto důvodu bylo provedeno měření rychlosti pro různě velká mračna na rozdílných strojích. Výsledky jsou shrnuty v grafu 16.



Obrázek 16: Srovnání rychlosti výběru bodu

Podle očekávání je rychlost výpočtu na méně výkonných sestavách pomalejší. Avšak pro výběr bodu u mračna s maximálním počtem bodů byl zapotřebí čas v řádu jednotek sekund. Vzhledem k očekávanému použití se nejedná o čas, který by uživatele významným způsobem omezoval. Pro zrychlení výpočtu sevřeného úhlu mezi vektory by šla dále provést paralelizace výpočtu. V případě čtyřjádrového procesoru se ovšem vzhledem k lineární složitosti výpočtu nedá očekávat razantní nárůst rychlosti.

Rozdíl v prvních dvou sloupcích pro stejnou testovací sestavu lze přičíst na vrub množství nainstalovaných programů na OS Windows a prováděném multitaskingu. V případě OS Fedora šlo vždy o čistou instalaci.

8.2 Spotřeba paměti

Způsob vytváření OpenGL objektů a alokace paměti na grafické kartě je věc jejího ovladače. Z tohoto důvodu může celková pamět mezi voláními `glSwapBuffers()` kolísat. Podle wiki stránek ovladač karty často ve druhém vlákně data v paměti karty vhodně optimalizuje, komprimuje nebo v případě nedostatku paměti, kopíruje mezi pamětí počítače a vlastní pamětí. Odhadovaná potřebná paměť pro uložení bodů je zobrazena v tab. 5.

Pro uložení souřadnic jednotlivých bodů je v paměti počítače vytvořeno pole pro datový typ float. Každá souřadnice bodu potom odpovídá jedné hodnotě typu float, tzn. tři hodnoty pro každý bod. Pokud je v datovém souboru definována barva, je pro každou její složku alokována paměť typu float. Datový typ float byl zvolen z důvodu zpracování barvy ovladačem OpenGL, který předpokládá barevnou hodnotu v rozsahu 0 - 1. Celková velikost alokovaného pole pro barvy potom odpovídá velikosti pole pro uložení souřadnic bodů. Alokovaná paměť pro jeden bod je potom šest hodnot typu float.

Pokud je mračno bodů větší než 20000, je nezbytné alokovat další paměť pro pole indexů. Tyto indexy slouží k vykreslování pouze části bodů mračna. Velikost tohoto pole je 20000× velikost datového typu unsigned int. Pokud je program zkompilován pro 32 bitovou platformu, je nezbytné alokovat místo pro uložení jednoho bodu o velikosti $6 \times 4 = 24$ bytů. V případě 64 bitové platformy je to 48 bytů. K této paměti je nutné dále přičíst 80 kB pro pole s indexy pro každý soubor s mračnem bodů.

Při načítání datového souboru je z důvodu využívání vnitřní třídy pro načítání *m4_pcd* alokována další dočasná paměť. Proto je při načítání krátkodobě využito dvojnásobku potřebné paměti. V inicializační fázi už je veškerá pomocná paměť uvolněna a tedy není zahrnuta v tabulce odhadu spotřebované paměti.

Veškerá tato pamět je po inicializaci zkopírována na grafickou kartu pomocí volání OpenGL a dále se k ní nedá přistupovat. Program musí uchovávat souřadnice všech bodů mračna pro pozdější zpracování kvůli možnosti jednotlivé body vybírat. Z tohoto důvodu je celkový objem alokované paměti o 1,5 násobek vyšší (paměť pro souřadnice bodů, která je vždy v hlavní paměti). Pro jeden bod je nakonec zapotřebí paměť o velikosti 36 a v případě platformy x64 72 bytů.

Program pro svou funkci musí alokovat také paměť pro bitmapové textury. Jedná se o bitmapu s fontem a několik bitmap pro vyobrazení stavu kamery. Tyto bitmapy se v paměti ukládají v barevném formátu RGB s datovým typem float. Není zde uložený alfa kanál pro průhlednost. Průhlednost je dále řešena ve fragment shaderu na základě červené barvy v případě textu. Pro uložení jednoho texelu je zapotřebí 12, resp. 24 bytů na 32, resp. 64 bitové platformě. Po načtení bitmapových obrázků se paměť okamžitě přesouvá na grafickou kartu. Dohromady paměť pro textury zabírá kolem 1 MB.

počet bodů	paměť RAM	paměť GPU	sdílená paměť RAM
10 000	120 kB	240 kB	360 kB
30 000	360 kB	720 kB	1,1 MB
50 000	0,6 MB	1,2 MB	1,8 MB
100 000	1,2 MB	2,4 MB	3,6 MB
300 000	3,6 MB	7,2 MB	10,8 MB
500 000	6 MB	12 MB	18 MB
1 000 000	12 MB	24 MB	36 MB
3 000 000	36 MB	72 MB	108 MB
5 000 000	60 MB	120 MB	180 MB
10 000 000	120 MB	240 MB	360 MB
30 000 000	360 MB	720 MB	1,08 GB
50 000 000	600 MB	1,2 GB	1,8 GB
70 000 000	840 MB	1,7 GB	2,5 GB

Tabulka 5: Odhad potřebné paměti v závislosti na počtu bodů

8.3 Rychlost vykreslování

Pro testování výkonnosti programu bylo zvoleno porovnávání rychlosti vykreslování v závislosti na počtu bodů. Kvůli zdvojnásobení alokované paměti při načítání datového souboru není možno načítat jediný soubor s desítkami milionů bodů. Při načítání totiž není k dispozici dostatek paměti. Řešením pro testování výkonnosti bylo velké soubory rozdělit na menší s cca 1000000 body a využít možnosti načítání více souborů najednou. Pro testování výkonnosti byly použity sestavy popsané ve zbytku této kapitoly.

Sestava1

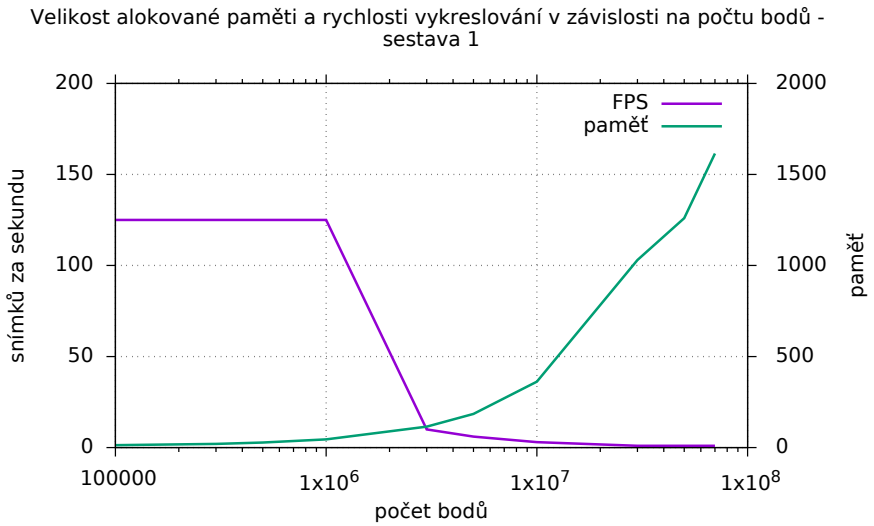
Intel CORE i5, Intel HD Graphics 4000, 4 GB RAM, OS Windows 10. Integrovaná grafická karta, která je součástí čipu Intel nemá vlastní paměť. Veškerá data jsou tedy uložena v hlavní paměti. Graf rychlosti vykreslování a spotřeby paměti lze vidět na obr. 17.

Sestava2

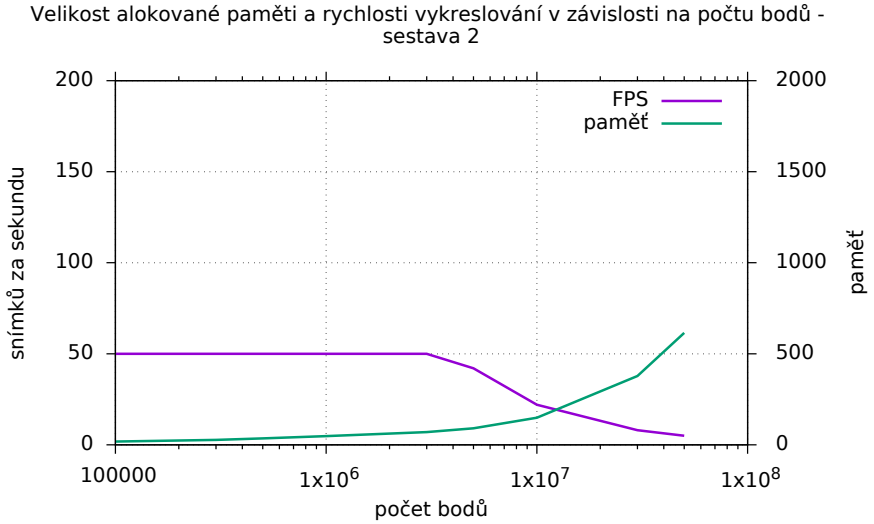
Intel CORE i5, GeForce GT 740M, 4 GB RAM, OS Windows 10. Sestava s touto grafickou kartou by podle výrobce měla být až 5x výkonnější než předchozí sestava. Grafická karta disponuje vlastní 2 GB pamětí typu DDR3/GDDR5. Graf rychlosti vykreslování a spotřeby paměti lze vidět na obr. 18.

Sestava3

Intel CORE i5, GeForce GT 740M, 4 GB RAM, OS Fedora 23. Jedná se o stejnou sestavu jako sestava 2, jenom s rozdílným ovladačem grafické karty a OS. Graf rychlosti vykreslování a spotřeby paměti lze vidět na obr. 19.



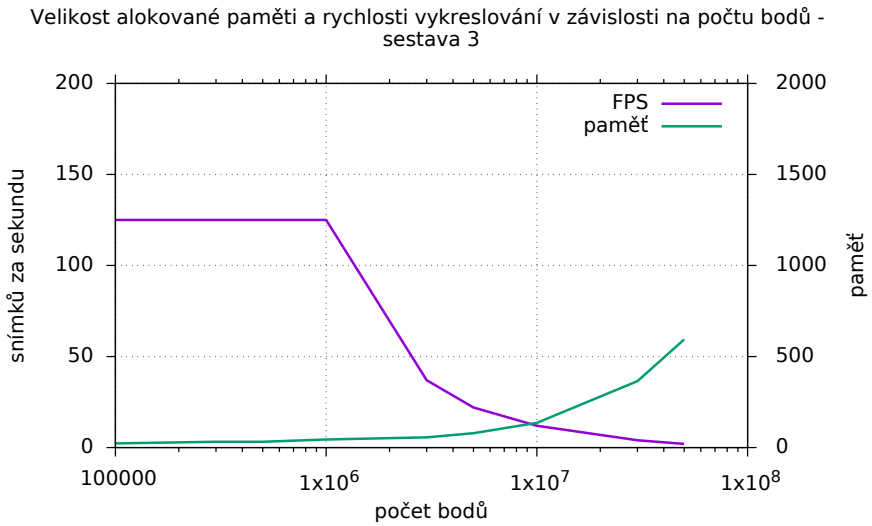
Obrázek 17: Rychlost vykreslování a množství potřebné paměti v závislosti na počtu bodů pro sestavu 1



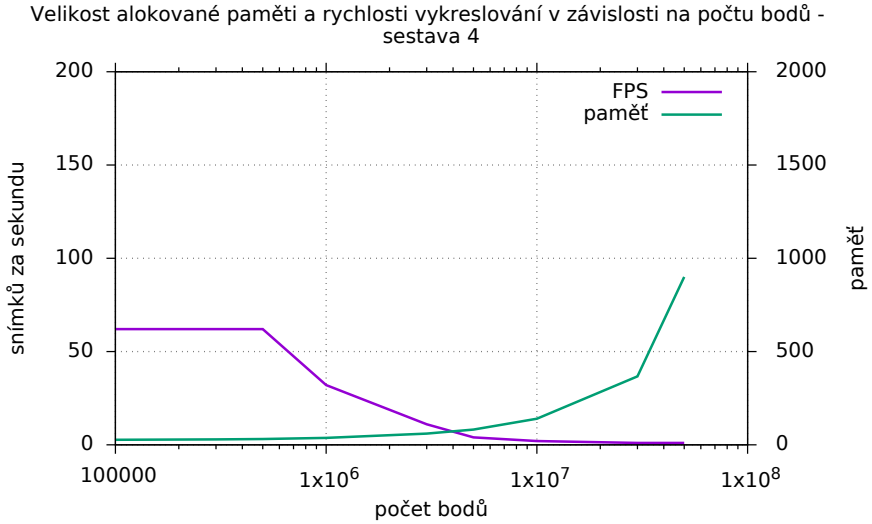
Obrázek 18: Rychlost vykreslování a množství potřebné paměti v závislosti na počtu bodů pro sestavu 2

Sestava4

Intel Pentium 4, nVidia G92, 2 GB RAM, OS Fedora 23. Jedná se o sestavu s poměrně starším jednojádrovým procesorem a grafickou kartou s 512MB paměti typu GDDR3. Graf rychlosti vykreslování a spotřeby paměti lze vidět na obr. 20.



Obrázek 19: Rychlost vykreslování a množství potřebné paměti v závislosti na počtu bodů pro sestavu 3



Obrázek 20: Rychlost vykreslování a množství potřebné paměti v závislosti na počtu bodů pro sestavu 4

8.3.1 Souhrn provedených testů

Na základě provedených testů lze dospět k závěru, že výkonnost programu není zdaleka tolik ovlivněna použitým procesorem, ale – dle očekávání – grafickou kartou. Pokud karta disponuje dostatkem vlastní paměti, lze dosáhnout plynulého zobrazování mračen s desítkami milionů bodů. Maximální velikost zobrazovaných dat dále limituje množství operační paměti a použitý operační systému. Zde je ale nutno podotknout, že program nebyl porovnáván na počítačích s *čistou* instalací systému.

9 Problémy při vývoji

Při řešení zadání a zachování nezávislosti na počítačové platformně jsem se samozřejmě setkal s některými problémy. Hlavním z nich bylo nekonzistentní chování externích knihoven, které mají zajistit určitou míru abstrakce a nezávislost na operačním systému, resp. okenním manažeru. Dále jsem zjistil rozdíly v předávání parametrů mezi jednotlivými interprety příkazového řádku, resp. které speciální znaky se nepředávají.

Při použití multiplatformní knihovny GLUT pro práci s okny a periferiemi počítače jsem musel vyřešit problém s nekonzistentním generováním událostí na OS Windows a Linux. Rozdílné chování nastává při pohybu myši se zmáčknutým tlačítkem a současném stlačení speciálních kláves ALT, Control nebo Shift na klávesnici počítače. Zatímco u MS Windows se všechny události dají odchytit pomocí zpětného volání ve funkci `glutMotionFunc()` (volání při pohybu myši) a funkce `glutGetModifiers()` (vrací bitovou masku stlačených speciálních kláves), u OS Linux je pro ten samý účel nutno použít zpětné volání funkce `glutMultiMotionFunc()`, která na OS Windows není generována vůbec. Problém nastává, pokud uživatel speciální klávesu nestiskl. V tomto případě se událost stlačení myši odchytává standardně ve zpětném volání funkce `glutMotionFunc()`. Tuto nekonzistenci jsem nahlásil jako bug do projektu FreeGLUT pod číslem 226. Prvním řešením této nekonzistence pro mne bylo projít zdrojový kód knihovny FreeGLUT, kde jsem objevil nedostatek při zpracování událostí ve smyčce zpráv v OS Linux. Při současném stisku myši a speciální klávesy X server generuje `XI_Motion` událost namísto `MotionNotify` události. Knihovna FreeGLUT první událost zpracuje, zavolá `glutMultiMotionFunc()` a vygeneruje událost `MotionNotify`, kterou zařadí na začátek fronty zpráv X serveru. V tomto okamžiku ale poslední vytvořená událost neobsahuje všechny informace o stlačených speciálních klávesách. Takto opravenou knihovnu by ale bylo potřeba při překladu programu znovu zkompilovat nebo přibalit k aplikaci.

Druhou možností, kterou jsem nakonec implementoval, bylo zachytit stlačení speciální klávesy ve volání `glutMultiMotionFunc()` a nastavit pomocnou booleovskou proměnnou. Tímto řešením jsem se vyhnul stejnému kódu na dvou místech v programu. Při zpracování volání `glutMotionFunc()` je jenom potřeba tuto proměnnou zkontrolovat a případně správně nastavit strukturu reprezentující stlačené speciální klávesy. Program si tak zachoval nezávislost na OS a současně nemusí být distribuována zkompilovaná knihovna spolu s programem.

Druhým problémem byla nefunkčnost programu Ibis PCD Viewer s některými ovladači OpenGL od NVidie od verze 319 na různých distribucích OS Linux. Konkrétně se jedná o bug https://bugs.launchpad.net/ubuntu/+source/nvidia-graphics-drivers-319/+bug/1248642. Chyba je ve špatném linkování knihoven mezi Mesa a NVidia a vypadnutím odkazu na knihovnu *pthreads*. Řešením je přilinkovat knihovnu pthreads v Makefile souboru a zavolat libovolnou funkci pro práci s vlákny v programu. Z tohoto důvodu je v souboru entry.cpp na začátku otestován OS a zavolána funkce `pthread_getconcurency()`.

10 Návrhy na další zlepšení

Program poskytuje základní zobrazení a navigaci ve virtuálním prostoru. Jako další funkcionalitu by bylo možno použít metody k filtraci odlehlých bodů nebo bodů, které byly špatně změřeny za pomoci segmentace. Tuto funkcionalitu nabízí soubor funkcí z projektu OpenNI. OpenNI nám dále nabízí automatické spojování (tzv. registraci) jednotlivých mračen.

Dalším rozšířením programu by mohla být přenositelnost na platformu s OS Android nebo Raspberry Pi, resp. embedded grafické čipy firmy ARM. Vzhledem k minimalistickému využití knihovních volání OpenGL by neměl být problém program spustit, a to zejména v režimu kompatibility. Hlavní překážkou v aktuální verzi je využití knihovny glew, která pro tyto platformy momentálně není přenositelná.

Pro zrychlení programu, a to zejména zrychlení výpočtu při výběru bodu, bych implementoval výběr bodu za využití framebufferu. Princip spočívá v tom, že každému bodu je přiřazena unikátní barva. Přidělenou barvou jsou body následně vykresleny do framebufferu. Framebuffer v tomto případě není dále vykreslen na obrazovku. Po kliknutí myší stačí přečíst barevnou hodnotu vybraného pixelu z některého z OpenGL bufferů(stencil, color, frame) a porovnat ji s barvou, která byla přiřazena bodům. Pokud by byly všechny body seřazeny podle barvy v hešovací tabulce, měla by být celá operace mnohem rychlejší. Toto řešení by zároveň vyřešilo problém, kdy body, které jsou blízko kamery a jsou vykresleny jako velké kolečko, nejdou vybrat jinak než přesným kliknutím doprostřed.

Díky implementaci funkcionality undo/redo by bylo dále možné program rozšířit o ukládání celkového stavu. Tento stav by bylo možné obnovit při spuštění programu bez parametrů. Pokud by se serializovaly struktura *Snapshot* a všechny objekty tříd *CommandBase*, zbývalo by už jen pozměnit chování programu po startu.

Program využívá třídu *PointData*, která sdružuje informace o jednotlivých vrcholech a jejich barvách. Při načítání datového souboru ve formátu *pcd* používá specializovanou třídu. Dalším krokem by mohlo být použití specializované třídy pro načítání lidarových dat nebo dat uložených v *obj* formátu.

11 Závěr

Cílem diplomové práce bylo vytvořit OpenGL prohlížeč naměřených prostorových dat. Tato data měla být reprezentována ve formě jednotlivých bodů, přímek a polygonů.

Celý program je postaven na OpenGL. Program je implementován pro PCD, resp. pro jejich datový formát *pcd*, ve kterém byla uložena naskenovaná data. Tento formát nepodporuje uložení informací o jednotlivých polygonech nebo přímkách. Z tohoto důvodu byl program tvořen primárně s cílem zobrazovat maximální množství bodů. Program dokáže načítat více oddělených souborů s měřeními. Jednotlivá naměřená data od sebe lze barevně oddělit. Program nad rámec zadání implementuje zavádění korekcí pro jednotlivá měření.

Hlavní důraz v práci měl být kladen na intuitivní a jednoduché ovládání pomocí klávesnice a myši. Pro jednoduché ovládání byl vytvořen koncept kamery, která snímá scénu. U kamery lze snadno měnit její parametry a pohybovat s ní v prostoru, aniž by uživatel musel cokoli složitě přepínat. Pro měření vzdáleností mezi body byl zvolen poměrně pomalý algoritmus, který je založen na operacích s vektory a jejich vzájemném porovnávání. Výpočet pro mnohamilionová data na méně výkonných procesorech může program výrazně zpomalit.

Požadavek na multiplatformnost a jednoduchou přenositelnost kódu byl splněn za pomoci využití prostředí OpenGL a dvou volně šířených knihovnen nezávislých na OS. Program byl úspěšně spuštěn na hlavních distribucích Linuxu a všech počítačích s MS Windows od verze XP. Program nedisponuje žádným uživatelským rozhraním. Základní informace jsou zobrazovány v informačním panelu a veškerá ostatní interakce probíhá za pomoci výpisů do konzole. Program zachovává podporu starších počítačů, na kterých nejsou aktuální ovladače OpenGL, za cenu zdvojení kódu při inicializaci a rozdílném vykreslování.

Během programování v OpenGL jsem se mnoho naučil – zejména matematickým základům nutným při každém programování ve 3D. Pro výběr bodu a vykreslování textu jsem přistoupil k vlastnímu řešení, které jistě není optimální, ale postačuje pro splnění zadání diplomové práce. Například vykreslování textu by se jistě nedalo použít u textového editoru napsaném v OpenGL, jelikož zde není podpora pro Unicode ani ligatury národní abecedy.

Práce splnila všechny požadavky, které byly součástí zadání. V závěrečných kapitolách práce je shrnuta výkonnost programu a nastíněn jeho další možný vývoj.

Literatura

[1] Dave Shreiner, The Official Guide to Learning OpengGL, Versions 3.0 and 3.1, 7. edice, Boston: Addison-Wesley, 2009, ISBN 978-0-321-55262-4

[2] Dave Shreiner, The Official Guide to Learning OpengGL, Versions 4.3, 8. edice, Boston: Addison-Wesley, 2013, ISBN 978-0-321-77303-6

[3] Richard S. Wright Jr, OpenGL super bible : comprehensive tutorial and reference, 5. edice, Boston: Addison-Wesley, 2010, ISBN 978-0-321-71261-5

[4] The PCD (Point Cloud Data) file format [online], [cit. 5. února 2016]. Dostupné z: <http://pointclouds.org/documentation/tutorials/pcd_file_format.php>.

[5] Tutorial 3 : Matrices [online], poslední revize 22. března 2016 [cit. 22. dubna 2016]. Dostupné z: <www.opengl-tutorial.org>.

[6] OpenGL Mathematics (GLM) [počítačová knihovna]. Ver. 0.9.6 [cit. 20. března 2016]. Dostupné z: <www.glm.g-truc.net>. Soubor hlavičkových souborů.

[7] The OpenGL Extension Wrangler Library (GLEW) [počítačová knihovna].Ver. 1.11.0 [cit. 20. března 2016]. Dostupné z: <http://glew.sourceforge.net/>. Předkompilovaná knihovna pro Windows.

[8] The freeglut project [počítačová knihovna].Ver. 3.0.0 [cit. 20. března 2016]. Dostupné z: <http://freeglut.sourceforge.net/>. Předkompilovaná knihovna pro Windows.

[9] The Mesa 3D Graphics Library [počítačová knihovna], Ver. 11.0.1 [cit. 20. března 2016]. Dostupné z: <www.mesa3d.org>.

[10] History of OpenGL [online], poslední revize 18. ledna 2016 [cit. 20. ledna 2016]. Dostupné z: <https://www.opengl.org/wiki/History_of_OpenGL>.

[11] OpenGL® Registry [online], poslední revize 20. dubna 2016 [cit. 20. března 2016]. Dostupné z: <www.opengl.org/registry>.

[12] Codehead's Bitmap Font Generator [počítačový program], Ver. 1.43. [cit. 20. března 2016]. Dostupné z: <http://www.codehead.co.uk/cbfg/>.

[13] Vizualizace dat, Scalar fields visualization [elektronické prezentace], poslední revize 8. října 2015 [cit. 20. března 2016]. Dostupné z: <http://mrl.cs.vsb.cz/people/fabian/vd_course.html>.

A Obsah přiloženého DVD

Na přiloženém DVD se nachází následující adresářová struktura:

- *fonts/* – složka s bitmapou a metadaty fontu použitým v programu.
- *img/* – složka s obrázky, které symbolizují ovládání kamery.
- *include/* – složka s hlavičkovými soubory.
- *libraries/* – složka s hlavičkovými soubory knihoven FreeGLUT a GLEW. Složka dále obsahuje dynamické knihovny pro spuštění programu pod OS Windows.
- *pcd/* – složka s testovacími daty.
- *projects/* – složka s projekty programu.
  - *visual studio/* – složka s projektem Visual Studia pro OS Windows včetně spustitelného programu.
  - *make/* – složka s Makefile souborem pro OS Linux.
- *shaders/* – složka s naprogramovanými shadery v jazyce GLSL.
- *source/* – složka se zdrojovými kódy programu.
- *readme.txt* – textový readme soubor s popisem spuštění a ovládání programu.
- *readme.html* – html readme soubor s popisem spuštění a ovládání programu.

B Výpisy programovatelných shaderů

```
#version 130

in vec3 vertices_in;
in vec3 color_in;
in vec2 uv_in;

uniform mat4 MVP;
uniform vec4 uniform_color;
uniform bool uniform_is_color_set;
uniform bool uniform_texture_is_set;
uniform int texture_id;
uniform bool uniform_font_drawing;
uniform float uniform_point_size;
uniform bool use_point_parameters;

out vec4 f_color_in;
out vec2 f_uv_in;

void main()
{

   gl_Position = MVP * vec4(vertices_in, 1.0);

   if(use_point_parameters) {
      float d = length(gl_Position);
      gl_PointSize = uniform_point_size * (1.0f/(0.24f * d));
   }
   else
      gl_PointSize = uniform_point_size;

   if(gl_PointSize < 0.1f)
      gl_PointSize = 0.1f;
   if(gl_PointSize > 20.0f)
      gl_PointSize = 20.0f;

   f_color_in = vec4(color_in, 1.0);
   f_uv_in = uv_in;
```

```
38
39 }
```

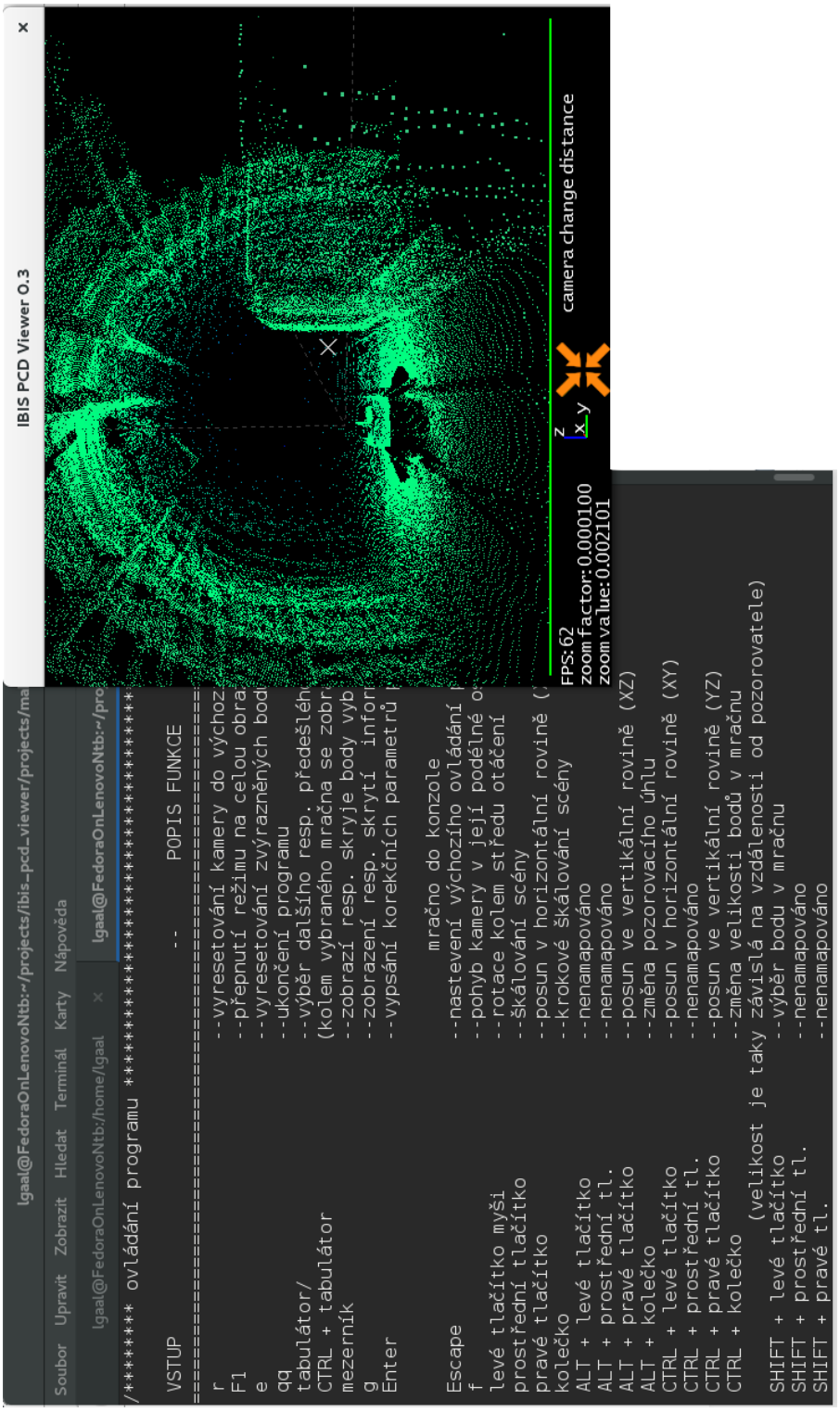
Výpis 8: Ukázka použitého vertex shaderu

```
1  #version 130
2
3  in  vec2 f_uv_in;
4  in  vec4 f_color_in;
5
6  uniform vec4 uniform_color;
7  uniform bool uniform_is_color_set;
8  uniform bool uniform_texture_is_set;
9  uniform sampler2D myTextureSampler;
10 uniform bool uniform_font_drawing;
11
12 out vec4 color_out;
13
14 void main()
15 {
16    if(uniform_font_drawing)
17    {
18       color_out = texture(myTextureSampler, f_uv_in);
19       if(color_out.r < 0.2f)
20          color_out.a = 0.0f;
21    }
22    else if(uniform_texture_is_set)
23    {
24       color_out = texture(myTextureSampler, f_uv_in); // vec4(1.0f, 0.0f, 0.0f,
             1.0f);
25       if(color_out.r > 0.7f && color_out.g > 0.7f && color_out.b > 0.7f)
26       {
27          color_out.a = 0.0f;
28          color_out.r = 0.0f;
29       }
30    }
31    else if(uniform_is_color_set)
32    {
33       color_out = uniform_color;
34    }
```

```
35     else
36     {
37        color_out = f_color_in;
38     }
39 }
```

Výpis 9: Ukázka použitého fragment shaderu

C Další ukázky programu



Obrázek 21: Výpis nápovědy do konzole

Obrázek 22: Mnohamilionové mračno s ukázkou měření vzdálenosti

D Ostatní tabulky

Vstup	Popis funkce
F1	přepnutí režimu na celou obrazovku
r	nastavení kamery do výchozí pozice, smazání vybraných bodů, vymazání undo/redo zásobníků
e	smazání zvýrazněných bodů
qq	ukončení programu
tab	výběr dalšího mračna
CTRL + tab	výběr předešlého mračna
mezerník	přepnutí viditelnosti vybraného mračna
g	zobrazení, resp. skrytí informačního panelu
enter	vypsání korekčních parametrů aktuálně vybraného mračna do konzole
escape	nastavení výchozího ovládání programu
f	pohyb kamery v její podélné ose
levé tl. myši	rotace kolem středu
prostřední tl. myši	plynulé škálování scény
pravé tl. myši	posun v horizontální rovině (XY)
kolečko	krokové škálování scény
ALT + pravé. tl.	posun ve vertikální rovině (XZ)
ALT + kolečko	změna pozorovacího úhlu kamery
CTRL + pravé. tl.	posun ve vertikální rovině (YZ)
CTRL + kolečko	změna velikosti vykreslovaného bodu
SHIFT + levé tl.	výběr bodu v mračnu
SHIFT + kolečko	změna parametru rychlosti škálování (slouží k jemnému doladění přiblížení scény)
CTRL + SHIFT + levé tl.	změna vzdálenosti kamery od středu otáčení
x + pravé tl.	posun kamery v ose X
y + pravé tl.	posun kamery v ose Y
z + pravé tl.	posun kamery v ose Z
X + pravé tl.	posun vybraného mračna v ose X
Y + pravé tl.	posun vybraného mračna v ose Y
Z + pravé tl.	posun vybraného mračna v ose Z
A + pravé tl.	rotace vybraného mračna kolem osy X
B + pravé tl.	rotace vybraného mračna kolem osy Y
G + pravé tl.	rotace vybraného mračna kolem osy Z
CTRL + U	krok v historii zpět
CTRL + R	krok v historii vpřed

Tabulka 6: Ovládání programu Ibis PCD Viewer